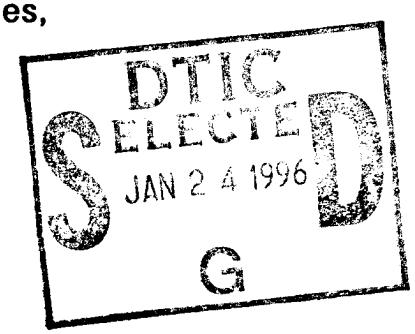
RL-TR-95-103
Final Technical Report
June 1995

REAL-TIME EMBEDDED HIGH PERFORMANCE COMPUTING: APPLICATION BENCHMARKS

The MITRE Corporation

Curtis P. Brown, Mark I. Flanzbaum, Richard A. Games,
and John D. Ramsdell



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-103 has been reviewed and is approved for publication.

APPROVED: Paul Sierak

PAUL SIERAK
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Final Oct 93 - Oct 94

4. TITLE AND SUBTITLE
REAL-TIME EMBEDDED HIGH PERFORMANCE COMPUTING: APPLICATION BENCHMARKS

5. FUNDING NUMBERS
C - F19628-94-C-0001
PE - 62702F
PR - MOIE
TA - 74
WU - 11

6. AUTHOR(S)
Curtis P. Brown, Mark I. Flanzbaum, Richard A. Games, and John D. Ramsdell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The MITRE Corporation
Mail Stop E025
202 Burlington Road
Bedford MA 01730

8. PERFORMING ORGANIZATION REPORT NUMBER
MTR 94B0000145

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-95-103

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Paul Sierak/C3CB/(315) 330-4065

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The final report develops realistic benchmarks to assess the applicability of current memory massively parallel processors (MPPs) for real-time embedded applications, such as synthetic aperture radar (SAR) and space-time adaptive processing (STAP). A scalable real-time mapping of a generic two-dimensional processing chain applicable to SAR and STAP is developed and analyzed.

14. SUBJECT TERMS
Parallel processing, Real-time, Embedded, Benchmarks

15. NUMBER OF PAGES
80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

ABSTRACT

This paper develops realistic benchmarks to assess the applicability of current distributed memory massively parallel processors (MPPs) for real-time embedded applications such as synthetic aperture radar (SAR) processing and space-time adaptive processing (STAP). The benchmarks are applied to the Intel Paragon in anticipation of the availability of the Embedded Touchstone. A benchmarking methodology is developed that assesses the level of real-time performance provided by the current hardware/software system by determining the minimum period that can be sustained for a selected processing and/or communication function. Optimized library routines allow efficient processing at the nodes. Improving NX message-passing primitives and SUNMOS allow efficient communication on the backplane. Processing efficiencies are maintained for pipeline processing in which double buffering is used to largely hide the overhead of interprocessor communication. The real-time scalability of parallel implementations is considered. A scalable real-time mapping of a generic two-dimensional processing chain applicable to SAR and STAP applications is developed and analyzed.

PREFACE

This is one of three MITRE Technical Reports documenting work performed during Fiscal Year 1994 on MITRE Project 74110, Real-Time Embedded High Performance Computing. The complementary reports are:

1. *Parallel Implementation of the Planar Subarray Processing Algorithm*, MTR 94B114, by Richard A. Games and Dan S. Pyrik, and

2. *Real-Time Embedded High Performance Computing: Communications Scheduling*, MTR 94B146, by Richard A. Games, Arkady Kanevsky, Peter C. Krupp, and Leonard G. Monk.

ACKNOWLEDGMENTS

This work was supported by the United States Air Force Electronic Systems Center and performed under MITRE Mission Oriented Investigation and Experimentation (MOIE) Project 74110 of contract F19628-94-C-0001, managed by Rome Laboratory/C3CB. This work was supported in part by a grant of HPC time from the DOD HPC Major Shared Resource Center, Wright-Patterson Air Force Base. We wish to thank the Honeywell Corporation, Space Systems, Clearwater, Florida, for the use of their Paragon and for their assistance. Thanks to David Scott and other Intel software engineers for their assistance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

This paper develops realistic benchmarks to assess the applicability of current distributed memory massively parallel processors (MPPs) for real-time embedded applications. If the computational requirements of current and future real-time embedded systems can be satisfied by emerging MPPs, then costly application-specific processors and disparate data processors can potentially be replaced by a single homogeneous, scalable, programmable computing platform that is designed to track the progression of commercial technology.

1.1 BACKGROUND

The ARPA-Honeywell Embedded Touchstone program (Blitzer, 1993) is producing an embeddable high performance computer that incorporates commercial microprocessors running the same system and application software as its commercial counterpart, the Intel Paragon. This paper examines the performance provided by the commercial Paragon from the viewpoint of motivating real-time embedded applications. These applications, e.g., sensor system processing, have requirements not found in large-scale scientific computing such as the need for real-time processing, multi-level security, and fault tolerance. In this paper we focus on developing software satisfying *hard* real-time requirements—results must be computed within strict deadlines or they lose their value, potentially causing catastrophic system failure.

Specialized system software is required for real-time processing. A companion report (Games, et al., 1994b) discusses the issues involved, especially for the problem of providing real-time guarantees on the communication network connecting the processors. Real-time system software for the Embedded Touchstone is currently under development by Honeywell and the Open Software Foundation-Research Institute (OSF-RI). Because of the current lack of real-time system software for the Paragon, we restrict out attention to signal processing applications, which have more structured real-time requirements. The results of this paper are

directly applicable to two current applications: synthetic aperture radar (SAR) processing and space-time adaptive processing (STAP).

Signal processing algorithms are often characterized by static, predetermined data flows. Problem instances are repeatedly presented in sequence with a fixed period, and the result for any given instance must be computed after its arrival by a fixed deadline. In these applications, high sustained processing and communication rates are needed to reduce the size, weight, and power requirements of the embedded processor. As a rule of thumb, we would like to attain at least 50% of the advertised peak processing rates.

1.2 SUMMARY OF RESULTS AND REPORT ORGANIZATION

We describe a benchmarking methodology that assesses the level of real-time performance available from today's MPPs for the periodic applications under consideration. The metric measured in these benchmarks is the minimum period that can be sustained for the selected processing and/or communication function under test. These performance results establish a baseline against which the progress of future MPP hardware and system software upgrades can be measured. Processing and communication kernels are benchmarked separately as part of a real-time parallel software development process. Integrated processing chains containing both processing and communication are configured to establish the possible end-to-end efficiencies for the application. A more detailed summary of each section follows.

Section 2 reviews fundamental concepts used in this report: latency and throughput, vector pipeline computing, the cache constraint, tuning an algorithm to a processor architecture, and benchmarking.

Section 3 focuses on the fast Fourier transform (FFT) as a representative and important signal processing function to evaluate the performance currently available at the processing nodes. The performance of vendor supplied optimized library routines for the FFT is assessed for the i860 microprocessor. Performance problems for short length FFTs are noted, and an assembly language optimization is developed to improve the performance in the situation

when multiple short FFTs need to be computed. These techniques are applied to create an efficient custom library call for the processing kernel of a new multistage procedure for forming SAR images (Perry, et al., 1994a,b). The processing kernel, called the foldfft, combines a short FFT with memory management and other vector operations. The implementation of this kernel illustrates a software development progression that trades off ease of programming and portability with the attainment of a desired level of performance. The customized library call's processing rate is seven times faster than the original straight C-code implementation.

Section 4 assesses the current message passing performance available on the Paragon. A simple single pair benchmark illustrates the progression in the Paragon's message passing performance that we were able to obtain as the operating system changed over the last year (24 to 75 Megabytes (MB)/second for the NX system, 154 MB/s for SUNMOS on 1/4 MB messages). A more complicated benchmark involving many simultaneously communicating pairs assesses the impact of link contention in more realistic message-passing patterns. This benchmark illustrates how the locally fair merge on the Paragon backplane makes the message passing performance of distant nodes dependent on the behavior of intermediate nodes. This "failure of fairness" phenomenon motivates the need to schedule the backplane to assure real-time communication performance (Games, et al., 1994a). Clock benchmarks catalog current operating system dropout behavior and mark the transition in the paper to periodic real-time benchmarking. The distributed Hartstone benchmark is used to measure real-time communication performance for synchronized dataflow. A programmable message passing technique is developed to assess more complicated periodic patterns. As an example, the performance of "corner turning on a line," which is a component of a scalable real-time mapping introduced in Section 6, is assessed. For this complicated pattern, the NX message passing primitives perform near their peak, but SUNMOS does not do very well at all, perhaps because of a lack of synchronization in the benchmark.

Section 5 examines pipeline configurations of relevant processing and communication kernels on the Paragon to assess the degree to which high processing efficiencies can be maintained in end-to-end implementations. This section demonstrates the use of double buffering to overlap communication and computation in pipeline processing. A testbench is

constructed on the Paragon with a source node (or nodes) that injects data into the processing chain under test at a prescribed rate and a sink node (or nodes) that collects and checks the output. Single node benchmarks with and without data communication of the FFT and/or corner-turning (matrix transpose) functions are used to assess the degree to which the communication latency can be hidden. The current buffering and control flow arrangement hides between 2/3 to nearly all of the communications latency, depending on the function under test. A two-dimensional FFT processing chain, which is applicable to SAR image formation, is tested. As expected the throughput of the pipeline is determined by the bottleneck node, i.e., the processing node with the largest minimum period. There is no further degradation in the minimum period as a result of the longer pipeline. The two FFT processing nodes in this three-stage pipeline sustain 53 MFLOPS on multiple instances of a 512-point FFT.

Section 6 considers the scalability of parallel implementations from a real-time perspective. Two general mapping approaches that scale with problem size to maintain a fixed throughput requirement are described, but each has potential problems maintaining a fixed latency requirement, which is usually more difficult. A generic two-dimensional processing chain applicable to SAR and STAP applications is considered. A two-dimensional pipeline mapping is proposed that can be scaled to implement the processing when one of the problem dimensions increases. The real-time scalability of this family of mappings is analyzed in terms of its ability to meet both throughput *and* latency requirements. The limiting component is a communication step involved in transposing a distributed matrix between the two processing stages. A recursive implementation of this so-called "corner turn on a line" is proposed and benchmarked using the techniques of Section 4. The required minimum period for this step grows sublinearly, implying that the proposed family of mappings can most likely be configured to meet the real-time requirements over practical ranges of input parameters.

Section 7 summarizes our conclusions and suggests areas of future work.

SECTION 2

FUNDAMENTAL CONCEPTS

This section reviews some fundamental concepts used in this paper. It uses the Intel i860 microprocessor for illustration.

2.1 LATENCY AND THROUGHPUT

In many computing applications, the objective is to minimize the amount of time required to produce a solution. For a problem instance P, if the inputs become available to the processor at time $t_{initial}$ and the solution is completed at time t_{final}, then the *processing latency* for that problem instance is given by $t_{final} - t_{initial}$.

Often in applications like sensor processing, a stream of problem instances $P_1, P_2, \ldots, P_i, \ldots$ must be processed. If there is a fixed time period p between each problem instance, then the problem *throughput* is given by $1/p$. Throughput is a rate, and in applications that are largely computational it is customary to express it in terms of the number of operations per second. For example, if a problem instance requires f floating point operations, then the throughput of $1/p$ problems per second implies that the processor must sustain f/p floating point operations per second (FLOPS).

In a real-time implementation, the application's throughput and latency requirements are specifications that the computing system must satisfy. In high throughput applications, the throughput requirement can have a much greater influence on a system's design. For example, a batch of output from a radar receiver must be consumed before the next batch arrives or else it will be lost. Often the processing latency can be increased to make it possible to meet a high problem throughput. For example, pipelining is a technique used to increase throughput at the expense of increasing latency. The additional latency is the result of more communication and buffering.

2.2 PIPELINED VECTOR COMPUTING

Notions of latency, throughput, and pipelining are equally applicable at the single processor level of a parallel implementation. To understand the limitations of vector processing on a pipelined microprocessor, such as the Paragon's i860 microprocessor, we need to examine the pipeline process and its relationship to the algorithm or vector process. Pipelined processing is a standard technique that distributes the processes among multiple elements, where each element can process a subtask rapidly. The solution is not available until the data is passed through all the elements in the chain (see Figure 2.1).

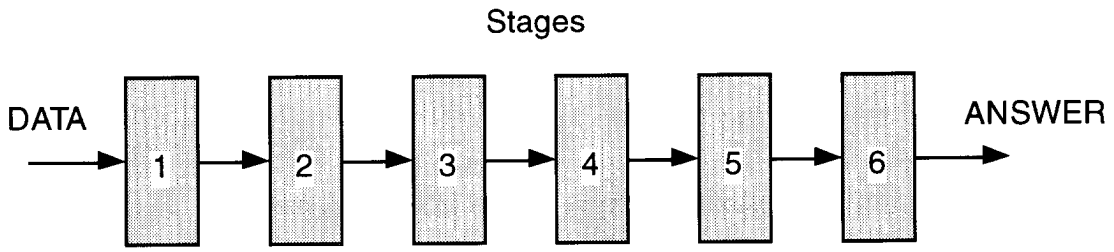


Figure 2.1. Pipeline Processing

The i860 floating-point arithmetic processor is actually composed of both a multiplier and an accumulator, with each having three stages of pipelining. Under steady state vector conditions the i860XP can perform a single precision multiply and addition in parallel at 50 million times per second, corresponding to a peak rate of 100 million FLOPS (MFLOPS). To start a pipelined vector operation we must first flush the pipeline to initialize the system to a known state, which in some cases requires as many as six additional cycles. Similarly, to complete a pipelined vector operation we must continue priming the pipeline, up to six cycles, to process the last element. Each time a pipelined vector operation is interrupted, computer cycles are wasted. This effect is typically referred to as a *pipeline stall.* As a result, operations on short vectors suffer from large amounts of overhead. However, efficient operation can be achieved if the data structures in the algorithm are designed to extend the effective vector length.

2.3 CACHE CONSTRAINT

The i860XP has a large but limited on-chip cache with a 128-bit bus width. High-speed vector processing can be performed at the full rate of the i860 clock on data that resides in the on-chip cache. Once the vector size exceeds the on-chip cache size, the processor is required to access the main memory through the smaller off-chip memory port (64 bits wide). In addition, the main memory access time is typically slower. As a consequence, the upper bound on the vector size that can be processed with maximum efficiency is fixed at approximately the cache size—the *cache constraint.* Typically most algorithms process multiple operands, further reducing the effective vector length and particularly the size of the data access stride (increment). Many hardware and software techniques exist to circumvent these limitations but they are typically time consuming to develop and expensive to support. Intel has inserted a number of hardware optimization techniques into the i860 to reduce the overhead; a discussion of the concepts can be found in (Intel, 1990, 1991, 1992a,b).

2.4 TUNING AN ALGORITHM TO THE COMPUTER ARCHITECTURE

A critical step in the implementation of an algorithm is the process of mapping the algorithm to the processor architecture. *Strip-mining* and *blocking* are two general optimization techniques used to extract the best performance offered from pipelined and cached systems. Operations on large vectors must be decomposed into vector operations that fit into cache yet are long enough to limit pipeline stalls. These partitioning techniques are typically referred to as strip-mining. At the other end of the spectrum, algorithms with small vector lengths not only pose a pipeline stall problem, but also present cache management issues. The goal of blocking techniques is to gather enough data with similar vector operations into the cache and process them with minimum pipeline stalls. By restructuring algorithms in this fashion a higher degree of processor utilization can be achieved. Our block FFT design in Section 3 illustrates the benefits of tuning an algorithm. A discussion of software optimization techniques can be found in (Zima and Chapman, 1990; Golub and Van Loan, 1991; and Dowd, 1993).

2.5 BENCHMARKING

The most accurate method for measuring system performance is to evaluate it on a fully implemented and stimulated system, but this measurement method is available only after the system is built. Project development guidance, control, and budgetary factors require the use of more timely methods for reducing risks and costs. One practical method is the use of well-formed benchmarks to model the performance behavior of the final system. But benchmarks, as with all reduced problem set models, yield performance predictions that are only as accurate as the benchmarks and their environment allow. A benchmark that is well-formed for one platform can become a less accurate model when the operating environment changes.

We are primarily interested in establishing how long a given computational or communication task will take. For embedded applications in which size, weight, and power constraints are paramount, the percent of utilization of the processor or communication link is also a crucial metric. As a result, it is typical to express benchmarking results in terms of processing or communications rates to facilitate comparison with the theoretical peak rates of the individual components of a system. We use the number of floating point operations to measure the computational complexity of the algorithms under consideration. This number is divided by the length of time of the calculation to determine the number of FLOPS. For a communication task, the number of bytes transmitted in the interval of time is expressed as bytes per second (B/s).

Parametrically exercising the benchmark is necessary to detect hidden dependencies. For example, for the FFT benchmark to be considered later, the transform size and the memory system can affect the performance. The benchmarks in this paper were conducted with two basic types of data management configurations, called *repeated instance* and *multiple instance*. A repeated instance benchmark repeats the processing on identical data. This type of test primarily exercises the program's ability to use the data cache and the processor architecture with minimal external memory dependencies. In the multiple instance case, as the name suggests, the algorithm operates on new data with each function execution. This exercises the cache-memory interaction and produces more realistic results.

In a real-time system the correctness of the result depends both on its value as well as when it is computed. A benchmarking methodology designed to assess a system's real-time performance is introduced in (Weiderman and Kamenoff, 1992). This approach incrementally loads the system and then assesses its ability to meet prescribed real-time constraints. In this paper we usually have a fixed processing load to perform, and we determine the minimum real-time period that can be guaranteed. The idea is to include into the benchmarking methodology those components of a real-time system that may impact ultimate performance, such as buffering and flow control. The predictive power of the benchmarks is crucial to their ultimate value to a parallel software engineering process that depends on knowing the capabilities of the MPP on the essential processing and communication kernels.

SECTION 3

FFT PROCESSING

In this section we describe our experiences with implementing an FFT on the i860 microprocessor. All the results assume single-precision floating point operations. The motivation for the work was a benchmarking study of the processing kernel of a new synthetic aperture radar image formation procedure (Perry, et al., 1994a,b). The so-called "foldfft" function used in this approach involves FFTs with short vector lengths, as short as length eight in actual applications. Pulse-doppler radar applications can also involve short FFT lengths. Unfortunately, the efficiency of vendor supplied library routines for such short lengths is low. In both these applications many short FFTs are required, and we develop a "block" FFT call to improve the processing efficiency in this case. We first describe the results of an FFT benchmarking study on the i860. Second, we describe the block FFT optimization and compare its performance with the existing call. Finally, we describe the application of the optimized call to computing the foldfft.

3.1 FFT PERFORMANCE ON THE i860

The exact execution time for a function is given by the number of computer cycles required to compute the function multiplied by the cycle period of the processor. Due to the complexity of the FFT algorithm and the new computer architectures, the number of computer cycles required is not easily computed. As a result, the execution time is typically measured from an actual implementation. In this work, we are interested in FFTs ranging from as small as eight points to as large as 8K points. Many facets of the i860 architecture can affect the performance of the FFT including: the pipeline, the memory hierarchy, the parallelism, the instruction set, and the clock speed.

The number of floating-point operations for the classical radix-two Cooley-Tukey FFT algorithm for N points is given by

$$10\frac{N}{2}\log_2(N). \tag{3-1}$$

This assumes 10 floating-point operations for each butterfly, $N / 2$ butterflies per stage, and $\log_2(N)$ stages required in the N point-FFT. Some of these operations involve ± 1 and $\pm i$, effectively reducing the operation count below that of expression (3-1). In addition, in some cases the FFT can be implemented with other radix bases that can reduce the operation count as well. However, in this paper we will uniformly use expression (3-1) for simplicity when we express FFT performance in terms of FLOPS.

Two assembly language FFT library calls were obtained for evaluation, one optimized for the i860XP (Kuck & Associates, 1993b) and the other optimized for the i860XR from Intel (Margulis, 1990). The source code for the Intel version was also available. The library routines were tested with the repeated instance and multiple instance benchmark procedures. Both configurations were tested for FFT lengths chosen from the sets {8, 16, 32, 64} and {512, 1024, 2048, 4094, 8192}. The benchmarks were run 14 October 1994 on a Paragon running OSF/1, revision 1.2.3, with its coprocessor enabled. All computations involve single precision, floating point numbers.

The results are presented in Figure 3.1. Only in the repeated instance case did we measure performance figures close to the maximum achievable. It is also apparent that the performance degrades for small and large transform sizes. In the multiple instance case, further degradation is observed due to constantly accessing external DRAM to obtain the new data used in each transform.

After close examination of the algorithm and the i860 architecture, two dominant effects were found to be causing the degradation. For the shorter transforms, the i860 floating-point pipeline is plagued with pipeline stalls. Within each FFT stage the pipeline must be primed for new data and flushed when completed. Since the i860 pipeline has six stages, this represents a significant overhead for transforms with 64 or less points. The second effect for the longer transforms is the cache size limitation. Once the data set does not fit into the

cache, the penalty for external memory accesses is additional memory cycles. The Kuck & Associates implementation evidently makes better use of the cache-memory interface for the longer transforms.

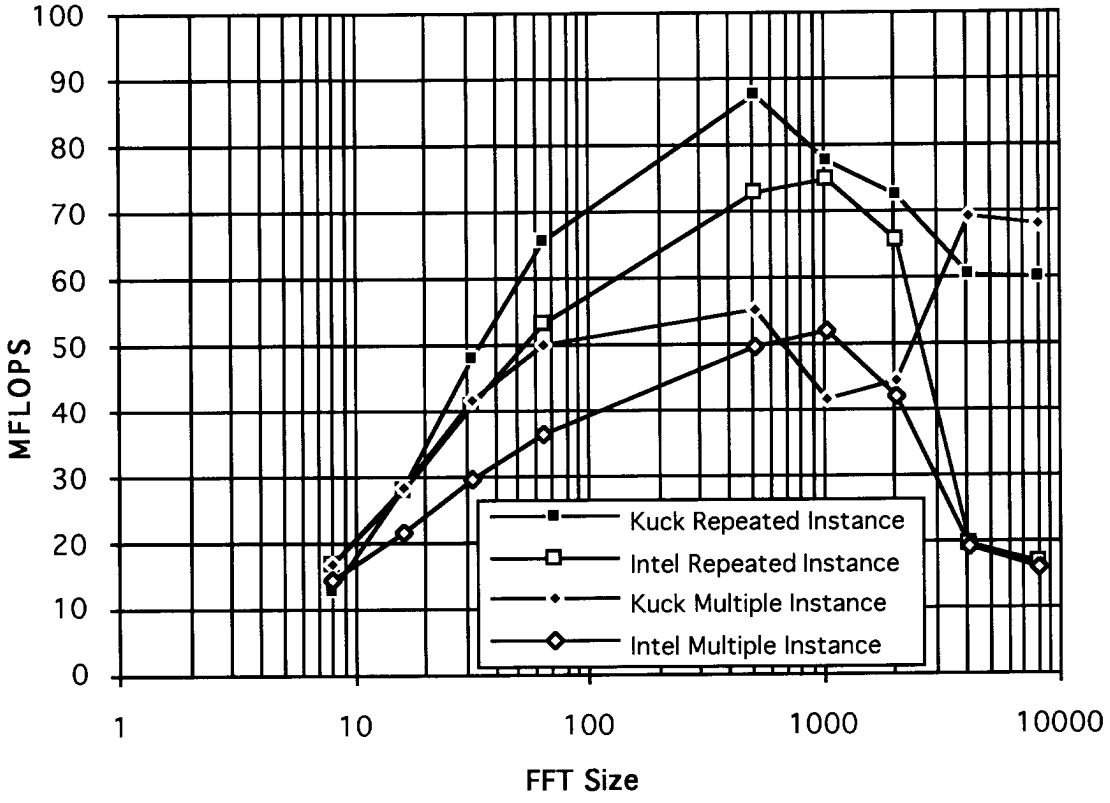


Figure 3.1. FFT Performance on the i860

It is interesting to note that for the Kuck & Associates FFT call the multiple instance benchmark performs better than the repeated instance benchmark for large FFT sizes (see the cross over between 2K and 4K in Figure 3.1). Since both benchmarks utilize the same library call, what accounts for this switch in performance? A number of subtle differences exist in the FFT implementation as the problem size increases that could cause this performance switch. One possible explanation involves the way the memory is accessed. The i860 processor has two types of memory access instructions—the FLD and PFLD instruction. The

FLD instruction is optimized for in-cache memory accesses and carries a significant time penalty for off-chip memory accesses. The PFLD instruction (pipelined floating point load) is optimized for off-chip memory access and has a time penalty associated with in-cache memory accesses. The programmer must be aware of the memory configuration to optimize the performance. The i860 will still operate properly if mismatched instructions are used, however at a reduced efficiency.

The Kuck & Associates FFT call employs both of these instructions depending on the memory utilization. The memory utilization is under control of the library routine for all of the FFT stages except for the first and last stage. Obviously the library call should assume the data is not in cache for the larger FFTs. Thus, the larger FFTs employ the PFLD instruction in the first stage. As a consequence the performance is slightly penalized for the repeated instance case when accessing in-cache data, which occurs about 25% of the time for the 4K case and 12% of the time for the 8K case. The discrepancy is further complicated by the way large FFTs are decomposed to fit in the cache to maintain good performance. As the FFT size grows, the decomposition /recomposition process overhead also plays a role in the performance figures.

3.2 BLOCK FFT DESIGN

The benchmark results in the last section clearly indicate that conventional optimization techniques employed in the FFT library routines suffer severe performance penalties as the FFT size is reduced. After close examination of the FFT algorithm, it was discovered that the pipeline stall problem for short input lengths could be addressed by restructuring the computations into a block format.

The signal flow graph for the decimation in frequency version of the FFT is shown in Figure 3.2 (Brigham, 1974). The i860 FFT algorithms process each stage of the FFT independently, thus stalling the pipeline after each stage. To eliminate this problem, the FFT algorithm was rewritten to extend the processing of each stage of the FFT across multiple data sets. The blocks are organized to fit in the cache independent of the FFT size. For example, for an FFT of length eight, we would load the data for 128 FFTs into cache (a 1024 complex word

cache) and process the data in stages. Stage 1 of all 128 FFTs would be completed before moving to stage 2, etc. In this arrangement, pipeline stalls would occur after 512 butterflies as opposed to four. The new arrangement allows the smaller FFTs to be implemented efficiently by trading a small amount of latency for throughput.

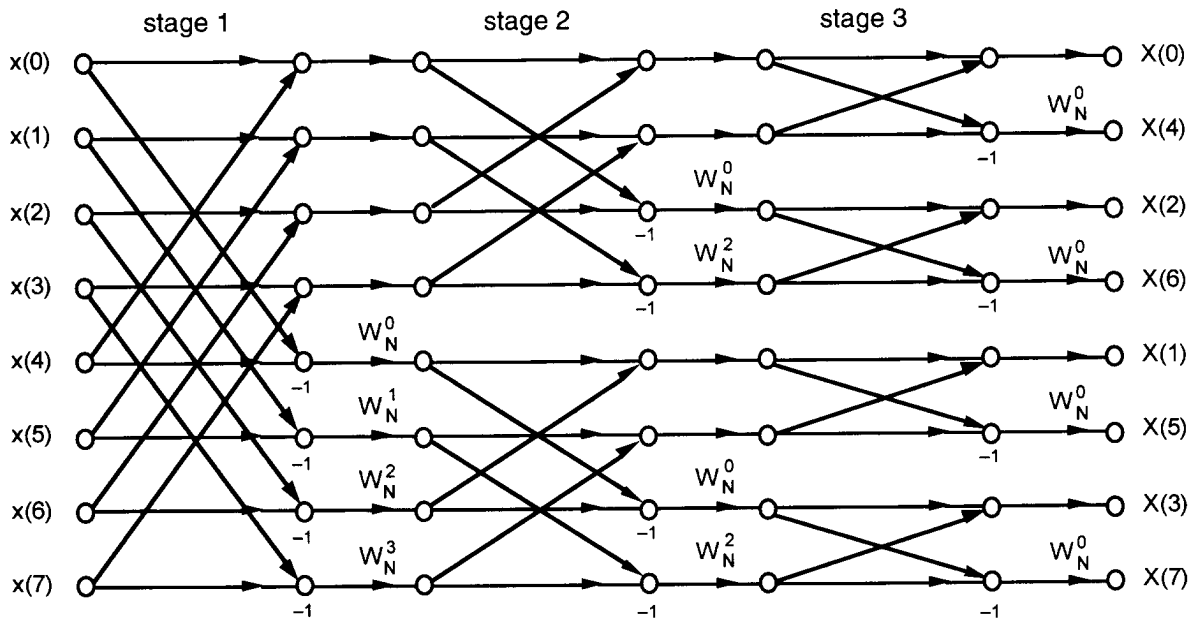


Figure 3.2. Signal Flow Diagram for a Length Eight Decimation in Frequency Radix-2 FFT

The block FFT was implemented by modifying the source code of the Intel library call. The source code for the more efficient Kuck & Associates call was not available. The benchmarking results for the blocked Intel call are shown in Figure 3.3. The performance improvement for the repeated instance case is dramatic. Each instance now computes multiple short FFTs on vectors contained in a block, but that block is repeated. For the multiple instance case, a 23 MFLOPS improvement is realized for 8-point FFTs. An improvement in performance continues for the larger vector lengths. The reason for the improvement is due to the cache-memory interface. In the FFT, the first and last stages are required to access external memory, causing cache misses. Since we are performing the FFT

in place, intermediate stages are able to take complete advantage of the on-chip cache. Thus, as the number of FFT stages is increased, the cache-memory interface becomes a smaller percentage of the cycle overhead.

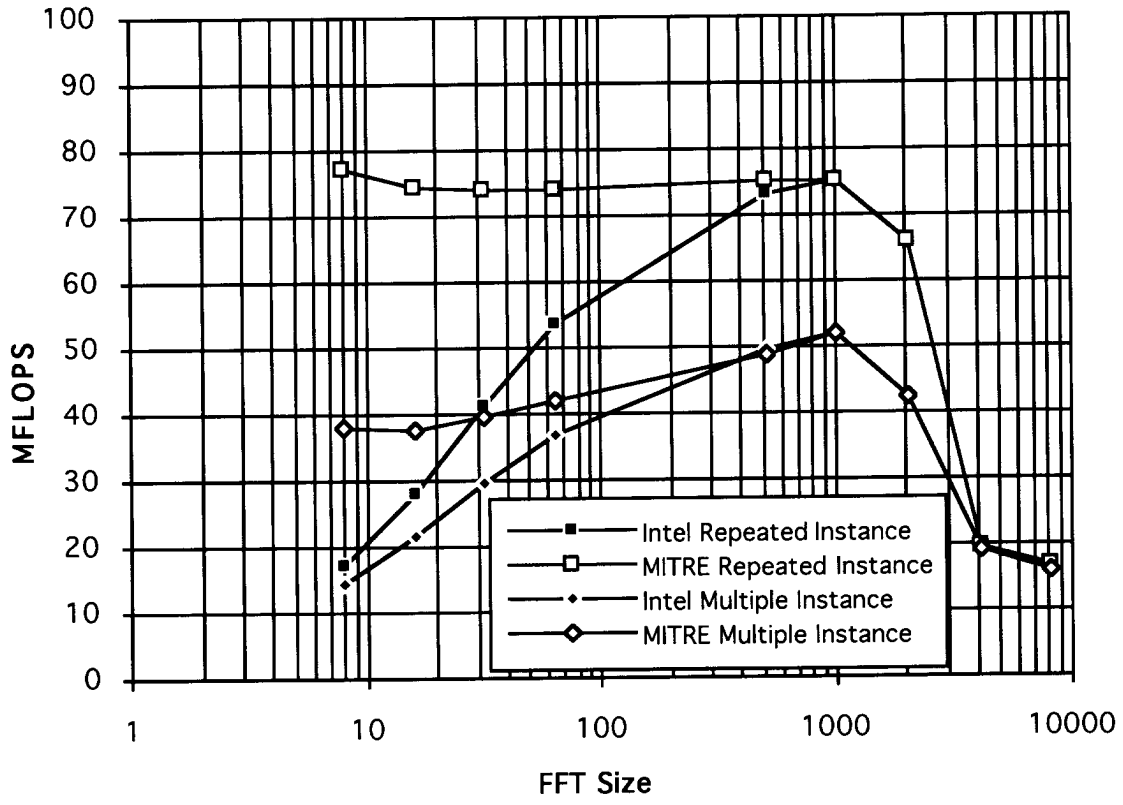


Figure 3.3. Complex FFT Performance on the i860XP with Multiple Instance Optimization

3.3 PLANAR SUBARRAY PROCESSING

A new algorithm for forming synthetic aperture radar (SAR) images, called Planar Subarray Processing (PSAP) was developed (Perry, et al., 1994a,b) to address the formidable computation and storage requirements implied by wide-area, high-resolution surveillance. PSAP has a number of features that make it suitable for efficient implementation on a distributed memory massively parallel processor (Games and Pyrik, 1994). The PSAP

algorithm consists of mostly independent and similar processing tasks whose organization permits a variety of possible parallelization strategies. However, this common processing kernel, called the foldfft, is more complicated than conventional kernels. A necessary condition for an efficient implementation of PSAP is that the foldfft can be implemented efficiently. This section applies the block optimization techniques described above to obtain an assembly language foldfft library call for the i860 that is seven times faster than a straight C-code implementation.

3.3.1 foldfft Processing Kernel

The foldfft function consists of a sequence of processing steps, including vector operations, array indexing, and an FFT, that makes it useful for comparing, or benchmarking, the performance available from current processor technology. Figure 3.4 shows a data flow representation and the mathematical description of the foldfft function. All the vectors are complex except for ***r***. Typical values for n and f in the PSAP algorithm include (64, 8), (128, 16), (256, 32), (16,16), (64, 64), and (512, 512), i.e., the length of the FFT involved can be quite short. In the symmetric case with $n = f$, the "fold to f" stage is eliminated and the foldfft reduces to a pre and post-weighted FFT. All the computations are assumed to involve single-precision, floating-point numbers. The number of floating-point operations for the algorithm is $2n + 2(n - f) + 5f\log_2 f + 6f$.

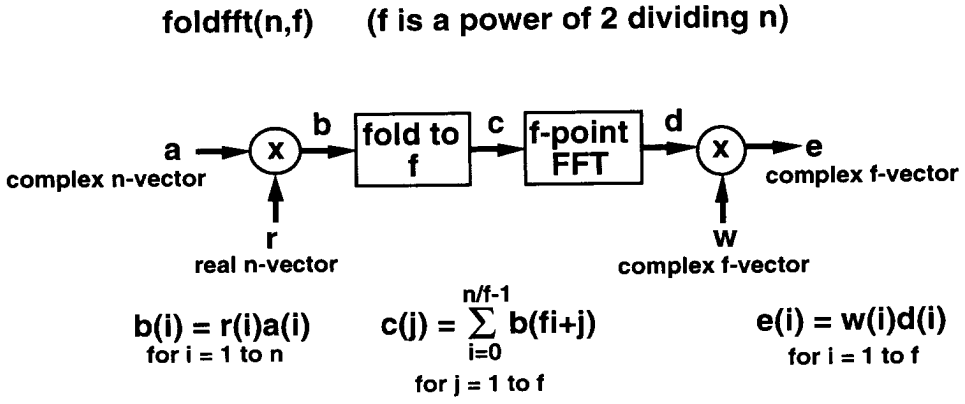


Figure 3.4. Dataflow Graph of the foldfft Function

3.3.2 Benchmarking the foldfft

Software implementations of the foldfft in "high level languages" have had limited success. We used a portable C-code implementation of the foldfft to test eight different workstation models and found that the Hewlett Packard HP 9000/735 workstation sustained the highest processing rate: 16.6 MFLOPS. The benchmark testing used a foldfft with $n = 64$ and $f = 8$. When the same code was run on the i860XR in a SKY Computer SKYbolt accelerator board, only 3.3 MFLOPS were measured. The performance improved significantly to 13.1 MFLOPS when SKYvec (SKY, 1993) library calls were included. These were repeated instance benchmarks, and so, somewhat optimistic for applications.

To improve the performance further in anticipation of a high-performance implementation, we coded the algorithm in assembly language. Previously developed optimizations techniques designed into the block FFT were applied to the foldfft. By restructuring the computations in the form of a block to fit within the cache and limiting pipeline stalls we dramatically improved the performance of the foldfft. Table 3.1 summarizes the previous benchmark results and the assembly language version for the i860XR processor on the Skybolt and the i860XP on the Paragon. The assembly language version provides almost a factor of 7 improvement over the C routines.

Table 3.1. foldfft(64,8) Benchmarking Results (MFLOPS)

Programming Approach	Skybolt i860XR	Paragon i860XP
C	3.3	6.8
C with Libraries	13.12	not done
i860 Assembly	44.4	47.3

The Paragon result corresponds to simply porting the i860XR code to the i860XP; the increase in performance is due to the faster clock, and even better performance would be expected if the code was modified to take advantage of the additional XP features.

3.4 CONCLUSION

This section examined the performance of the FFT on the i860 microprocessor. Vendor-supplied FFT library calls yielded efficient processing on realistic multiple instance benchmarks as long as the FFT size allowed efficient use of the cache. We showed that the efficiency for multiple short FFTs (sizes 8 through 128) could be improved by incorporating blocking into the FFT library call. These blocking ideas were applied to optimize the performance of the foldfft processing kernel. The foldfft is the basic building block of a new SAR image formation procedure that significantly limits the sizes of the FFTs involved in the processing.

Three different programming approaches were used in the foldfft study: plain C, C with optimized library calls, and a customized library call in assembly language. Each successive approach involves more programmer effort and is less easily ported to other platforms, with the more involved approaches providing increased performance (from 7 to 47 MFLOPS for Paragon processing). Obtaining high efficiency on the critical processing kernels is a necessary condition for applications that have strict size, weight, and power requirements. Maintaining such efficiencies when the processing kernels are combined into a parallel implementation depends on a variety of factors, including the performance of the underlying communication network. We examine the performance of the Paragon network in the next section and then return in Section 5 to the problem of sustaining high efficiencies in concert.

SECTION 4

COMMUNICATION AND CLOCK BENCHMARKS

In this paper, we are considering programs that deliver high performance on MPPs using some form of explicit message passing. These programs must surmount several barriers to be able to exploit the full bandwidth of the underlying message passing hardware. For example, care must be taken to minimize the number of times a message is copied. This section describes our experience using the Paragon OSF/1 operating system with the Intel NX message passing primitives. We also tested the Sandia-University of New Mexico Operating System (SUNMOS) (cf. Wheat et al., 1994). SUNMOS is a "lightweight" alternative to NX that eliminates certain features (packets, flow control) to obtain higher message passing rates.

We start with a simple point-to-point transmission without any contention, and gradually increase the complexity and contention involved in the message passing pattern, culminating in a pattern that implements a transpose operation or "corner turn" for a matrix that is distributed along a line of processors. We also introduce midway through the section a benchmarking methodology that assesses periodic real-time performance.

4.1 COMMUNICATION BENCHMARKS

The benchmarks in this section test message passing performance on a set of processors allocated in a single line. Unless otherwise noted, the benchmarks were run during September 1994 using the program `lin`, version 1.4 of 9 September 1994 on a Paragon running OSF/1, revision 1.2.3, with its coprocessor enabled. The program `lin` collected a number of previously separate benchmarks into a single framework to facilitate rerunning the benchmarks in the future. Again, unless otherwise noted, each node sent 1000 messages, each containing 262144 bytes ($262144 = 2^{18}$ or 1/4 MB). OSF/1 sends messages in packets of size 1792 bytes by default. The default size is the largest packet size supported by OSF/1.

The benchmark runs were initiated with the option `-plk` to make application pages ineligible for replacement by the OSF/1 virtual memory system. All message buffers were initialized to zero to force their pages to be resident in physical memory prior to the start of the benchmark. The nodes running a benchmark were synchronized by using the barrier synchronization procedure `gsync` before any timing began.

4.1.1 Adjacent Pair Benchmark

Purpose: The Adjacent Pair Benchmark measures the message transfer rate as a function of the message size and packet size of a single pair of processors communicating data in one direction.

Description: The Adjacent Pair Benchmark allocates two adjacent processors. One processor repeatedly transmits a message that is received by the other processor.

Results: The program `lin` was preceded by a program called `pair` that implemented just a few of the benchmarks in the `lin` program (the `pair` program sent twice as many messages as `lin` during each of its runs). The `pair` program was used to run the Adjacent Pair Benchmark at various times during the year using 1/4 MB messages and the default packet size. During that period, the OSF/1 operating system was upgraded from revision 1.1 to revision 1.2.3, and the message coprocessor was enabled. In June 1994, the SUNMOS operating system became available. The SUNMOS kernel provides no message flow control, but the original benchmark relied on the flow control mechanism provided by the NX message passing system in OSF/1. After adding code that explicitly acknowledged the receipt of a message by sending a message of zero length, the benchmark passed messages at the rate of 154 MB/s under SUNMOS. To our surprise, adding explicit acknowledgments to the benchmark improved the performance under OSF/1. Table 4.1 shows the progression of communication performance under OSF/1 over the course of the year.

Table 4.1. Adjacent Pair Benchmark for OSF/1

Date	Rate (MB/s)	Operating System	Comment
08 Dec 93	24	OSF/1 1.1	NX flow control
24 Jun 94	33	OSF/1 1.2.1	NX flow control
24 Jun 94	52	OSF/1 1.2.1	acknowledgments
29 Jul 94	86	OSF/1 1.2.1	acks and coprocessor
21 Sep 94	75	OSF/1 1.2.3	acks and coprocessor

As Table 4.1 indicates, enabling the message coprocessor provided substantial improvement in the performance of the benchmark run under OSF/1. The `pair` implementation of the benchmark ran at 86 MB/s and the `lin` implementation ran at 84 MB/s under OSF/1, revision 1.2. Due to a design problem, multiple i860s configured for symmetric multiprocessing can produce spurious writes to random locations unless the write-back cache is placed in so-called strong-order mode. All writes must execute in sequence in this mode, but allowing out-of-order writes can substantially improve performance. Revision 1.2.3 places the i860s in strong-order mode. We found the performance degradation for the `pair` version of the benchmark was about 15%.

Table 4.2 shows the communication performance as a function of message size; Table 4.3 shows the communication performance for 1000 1/4 Mbyte messages as a function of varying packet size.

Table 4.2. Adjacent Pair Benchmark with Varying Message Size

Message Size	262144	16384	1024	1
Rate (MB/s)	73.4	37.4	8.2	0.01

Table 4.3. Adjacent Pair Benchmark with Varying Packet Size

Packet Size	1792	1024	512	256	128
Rate (MB/s)	73.2	44.2	26.2	13.7	6.4

Each entry in Tables 4.1-4.3 gives a result measured for one run of the program. We ran the benchmark 10 times with the same parameters to assess the variability of the results. The case in which 1000 1/4 MB messages were received is typical of our measurements. In the group of 10 runs, the average throughput was 72.9 MB/s, the low was 71.9 MB/s, and the high was 73.6 MB/s.

The benchmark was modified so as to measure how the message transfer rate changes as a function of distance between a single pair of communicating nodes. Our results showed the throughput was independent of the separation distance.

4.1.2 Symmetric Many Pairs Benchmark

Purpose: The Symmetric Many Pairs Benchmark measures how the message transfer rate changes as a function of the number of communicating node pairs which use the same route.

Description: The Symmetric Many Pairs Benchmark allocates an even number of processors in a single line so that all message passing occurs in one dimension. The processors are grouped into pairs, one above the midpoint and one below it. Processors below the midpoint repeatedly send messages to their mates, which repeatedly receive messages. The layout for 16 processors is shown in Figure 4.1.

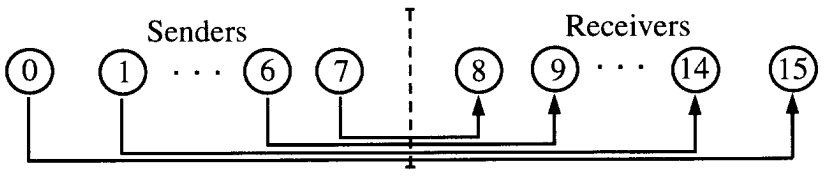


Figure 4.1. Symmetric Many Pairs Benchmark

In one version of the benchmark, the receipt of a message is acknowledged by sending a zero length message to the sender. A run terminates when any one pair completes the sending of a specified number of messages. That pair sends a termination message to all other pairs. Upon termination, the benchmark reports the data transfer rate for each receiver and the sum of those transfer rates, which is the rate data traveled through the backplane at the midpoint. The number of unsent messages for the pairs that are terminated prematurely are tallied.

Results: Table 4.4 shows the results of running the version of the Symmetric Many Pairs Benchmark using acknowledgments on a machine with the coprocessor turned on. The table shows the rate at which data passed through the middle communication link and the percentage of messages that were not sent due to the receipt of termination messages. This percentage gives a measure of the unfairness of the message passing. For the run using 16 processors, the rate for each pair is given in Table 4.5.

Table 4.4. Symmetric Many Pairs Benchmark Varying Number of Nodes

Processors	2	4	6	8	10	12	14	16
Sum (MB/s)	74.2	138.5	156.2	153.3	157.1	158.8	156.7	157.0
Unsent (%)	0	0	13	22	27	31	34	36

Table 4.5. Symmetric Many Pairs Benchmark with 16 Nodes

From - To	0-15	1-14	2-13	3-12	4-11	5-10	6-9	7-8	Sum
Rate (MB/s)	1.5	1.5	3.0	5.8	11.5	22.3	41.7	69.7	157.0

Discussion: The benchmark for 16 processors was run several times. While we did not statistically analyze the results of these runs, we observed roughly the same numbers for each run. In particular, with the exception of the first node pair, the throughput roughly doubles as the distance between nodes is decreased. As expected, the data entering the backplane from a

node is mixed packet-for-packet with the data traveling left-to-right. This locally fair merge leads to globally unbalanced throughput rates. This so-called "failure of fairness" presents problems for applications requiring predictable communication performance, since distant communicating pairs are subject to the communications behavior of intermediate nodes. This problem is treated in (Games, et al., 1994b), where protocols for "scheduling" the backplane are considered to insure predictable communication performance for real-time applications.

Before acknowledgments were added to the Symmetric Many Pairs benchmark, the benchmark would crash a Paragon when running OSF/1, revision 1.1. Apparently, the flow control mechanism failed with this benchmark.

4.1.3 Asymmetric Many Pairs Benchmark

This benchmark is identical to the Symmetric Many Pairs Benchmark except that the pairs are laid out so that the number of links between any two pairs is the same. The layout for 16 processors is shown in Figure 4.2. The results for 16 processors is shown in Table 4.6.

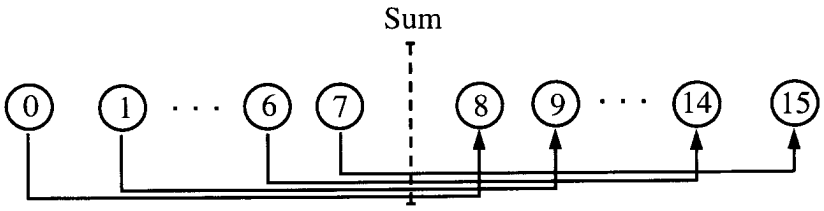


Figure 4.2. Asymmetric Many Pairs Benchmark

Table 4.6. Symmetric Many Pairs Benchmark with 16 Nodes

From->To	0-8	1-9	2-10	3-11	4-12	5-13	6-14	7-15	Sum
Rate (MB/s)	1.9	2.0	3.9	7.7	15.0	27.8	46.4	47.2	151.9

Discussion: Compared with the symmetric case, the overall throughput is only slightly reduced, but the throughput of messages sent from node 7 to node 15 is decreased. We suspect that the decrease is explained by the fact that packets sent from node 7 in the asymmetric case traverse a number of links, each of which is subject to contention, as compared to the symmetric case, which only involves a single link from node 7 to node 8.

4.2 CLOCK BENCHMARK

The benchmarks in this section time "drop outs" associated with the operating system. The benchmarks in the remaining sections of this paper are periodic real-time benchmarks. These benchmarks depend on performing tasks at regular intervals. However, the nonreal-time operating system currently preempts application tasks for varying intervals of time. The clock benchmark is intended to give some insight into the current state of affairs in this regard.

Purpose: The Clock Benchmark measures the pauses in program executions due to operating system overhead.

Description: The Clock Benchmark runs on one node. It zeros an array that is used to record the results of the experiment. It then reads the node's local clock using the `dclock` function repeatedly and computes the elapsed time since the last call to `dclock`. When the elapsed time is above a threshold, the time and the length of the pause is recorded in the array.

Results: The benchmark was run under OSF/1, revision 1.2.1, with the coprocessor turned on using the program `longclk`, version 1.2 on 24 June 1994. Figure 4.3 shows the results of a run lasting one half hour. There were 137 pauses of length greater than 500 μs. Most of these pauses have a duration of about 1 ms and occur at a regular rate of about once every 13 seconds. In addition, there were a small number of pauses in the 2 ms range.

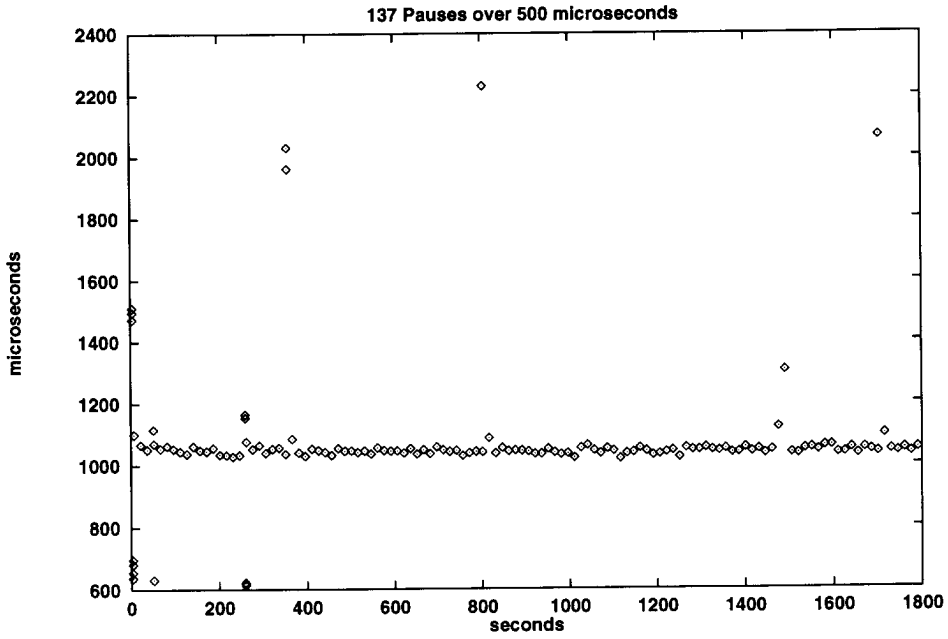


Figure 4.3. Clock Benchmark: Pauses > 500 μs

Figures 4.4 and 4.5 show the Clock Benchmark results for finer time scales. From Figure 4.4, we see that there were about 100 pauses of duration greater than 50 μs every second. Figure 4.4 also shows there is about a 200 μs pause every second and a slightly less than a 350 μs pause every two seconds that alternates with a slightly greater than a 350 μs pause. Figure 4.5 shows a one second interval and that the remaining pauses have a duration mostly in the range of 105–120 μs with a period of about 10 ms. We suspect that there is a timer interrupt every 10 ms to allow scheduling by the kernel.

Discussion: When the Clock Benchmark was run under SUNMOS, there were no pauses over 50 μs. An early version of this benchmark reported pauses with a duration of 5 ms. Zeroing the array that was used to collect the data eliminated those pauses. We concluded that the 5 ms pauses corresponded to the time it takes to page elements of the data array into physical memory. The early version of the benchmark was run specifying the page lock

runtime option (`-plk`), but this option simply keeps pages in memory once they have arrived. We could find no mechanism that forces an entire program into memory before it is executed. As a result of this experience, our benchmarks zero data arrays and message buffers before a timed test begins.

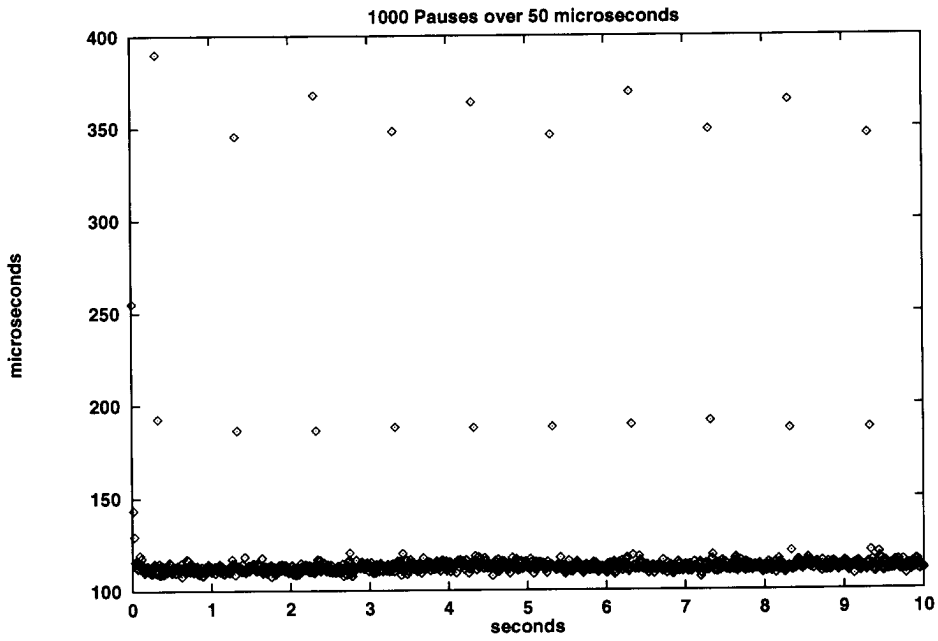


Figure 4.4. Clock Benchmark: Pauses > 50 μs

4.3 REAL-TIME BENCHMARKS

The benchmarks in this section are oriented toward real-time processing. They implement tasks that are executed at regular rates. Each execution of the task is expected to complete within a given period. The result reported for these benchmarks is the minimum period for which all deadlines are met.

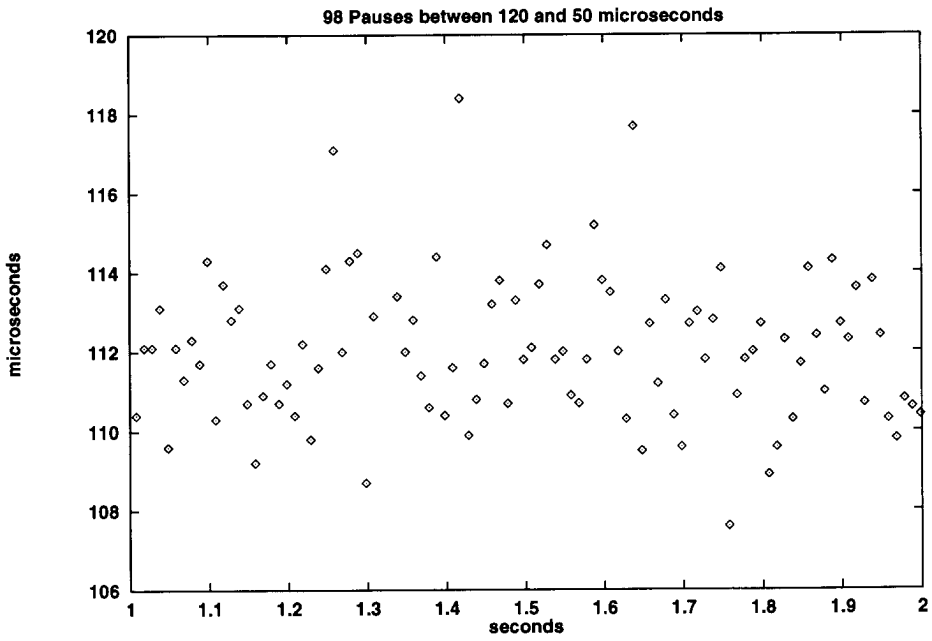


Figure 4.5. Clock Benchmark: Pauses Between 50 and 120 μs

4.3.1 Hartstone Distributed Benchmark

Purpose: The Hartstone Distributed Benchmark (Kamenoff and Weiderman, 1991) is one measure of how well a distributed computer handles real-time applications.

Description: The part of the Hartstone Distributed benchmark implemented uses two nodes of a distributed system. Each node executes *n* tasks. Each task on one node is paired with a task on the other node. One node contains only sending tasks and the other node contains receiving tasks. Future versions of the benchmark could add a processing workload at the sending and/or receiving node. There are no acknowledgments in this benchmark.

The input for the benchmark includes a period for each task pair. At the regular interval given by the period, a task on one node sends a message to its matching task on the other node. If the sender completes its transmission before the end of the period, it meets its deadline. The receiver meets its deadline if it completes the receipt of a message within the period.

The implementation of the benchmark takes advantage of the fact that the local node clocks on the Paragon are synchronized to within 1 μs. The sending node picks a time in the future at which the test will begin and then sends that time to the receiving node.

This implementation uses two nodes of a Paragon. There is one process on each node, and each process can spawn up to n POSIX threads, where $n = 3$ is the standard test configuration. An initial thread on each node spawns a separate thread for each task and waits for the completion of all tasks and then prints the results of the benchmark.

A thread implementing a task awaits the start of its period by reading the value of the local clock using `dclock`. If the period has yet to begin, the thread calls `pthread_yield`, which invokes the scheduler. The program does not attach priority to any particular thread, and no thread preempts another thread unless it consumes its time slice. The benchmark also has a compile-time switch that allows the program to be compiled so that there is only one task per node, and the POSIX thread library functions are not called.

Results: The benchmark was run under OSF/1, revision 1.2.3. The length of each message was 1/4 MB. The duration of the run was 1 second. When compiled so as to eliminate calls to the POSIX thread library, the smallest task period which met the deadlines was 4.2 ms and data was transferred at a rate of 62.4 MB/s. When the thread library was used, the smallest task period which met the deadlines was 5.0 ms and data was transferred at a rate of 52.4 MB/s. This degradation of .8 ms is evidently due to one call of `pthread_yield` in the loop checking `dclock`.

The benchmark was run with three tasks per processor, each passing 1/4 MB messages. The task periods were in the ratio of 1:2:4. The smallest period which allowed the benchmark to

meet all deadlines was 15 ms. Data was transferred between the nodes at a rate of 30.6 MB/s. Table 4.7 gives the individual rates for the three tasks.

Table 4.7. Distributed Hartstone Benchmark: $n = 3$

Period (ms)	15	30	60
Rate (MB/s)	17.5	8.7	4.4

Discussion: Every 60 ms all three threads have message passing tasks to perform. The current operating system shares the processor equally between the three threads, which puts the thread (task) with the shortest period clearly at a disadvantage. As a result, its minimum period more than triples compared to the single task case. Real-time operating systems often include priority based schedulers that should improve this situation. In the rate monotonic scheme, the priorities are assigned so that tasks with shorter periods have higher priority. Thus, at those points in the benchmark where all three tasks want to send messages, the task with the shortest period would have priority and monopolize the processor. There still would be the overhead associated with the potentially more complicated scheduling as well as multiple spin waiting loops. Indeed, when there are multiple threads, the spin waiting could have a significant impact as each thread is always busy. In the future, the overhead of an alarm-interrupt approach should be assessed.

4.3.2 Programmable Message Passing Benchmark

Purpose: The Programmable Message Passing Benchmark measures the real-time performance of a potentially complicated message passing pattern given as a simple program. It is a synchronized benchmark in that the periods are synchronized across all the nodes, and all send and receives complete within the common period.

Description: The Programmable Message Passing Benchmark is implemented using the program `pmp`. The program takes as input a message communication pattern and a period.

The benchmark tests if the message passing given by the pattern can be completed within the given period. The description of a pattern consists of a code sequence for each node. A code sequence specifies the sequence of actions to be performed by a node within a period. The following actions are currently supported:

1. **E**: Marks the end of a code sequence.

2. **R**: The node waits until a previously posted receive request returns a message, posts another receive request, and executes the next action in the code sequence. In this version of the benchmark, all receive requests are asynchronous.

3. **S** $\mathbf{N}_0$ $\mathbf{N}_1$: The node sends an $\mathbf{N}_1$ byte message to the node numbered $\mathbf{N}_0$ and executes the next action in the code sequence. In this version of the benchmark, all send requests are synchronous.

4. **W F**: The node waits until the fraction of the period given by **F** has passed and then executes the next action in the code sequence. **F** is a floating point number such that $\mathbf{F} < 1$.

5. **L N** $\boldsymbol{P}_0$: The node executes **N** times the code sequence denoted by $\boldsymbol{P}_0$ and then executes the next action in the code sequence. **N** is a nonnegative integer.

The Programmable Message Passing Benchmark is initiated by the following procedure. The message passing pattern and desired period is read by node 0, and the appropriate code sequence and common period are distributed to each node. For each node, the message buffers are zeroed and one receive request is posted. The processing on each node is synchronized by the use of the procedure `gsync`, after which each processor reads its local clock and then waits for one period. Finally, each processor repeats the following actions for a prespecified duration: it repeatedly calls `dclock` checking for the start of a period; at the beginning of each period it executes its code sequence; and it then checks to see if all processing was completed before the end of the period.

The benchmark was programmed to run the simple message passing pattern consisting of node 0 periodically sending a 1/4 MB message to node 1 and node 1 acknowledging each message with a zero length message. The code sequences at nodes 0 and 1 are respectively:

```
0: S 1 262144  R  E
1: R  S 0 0  E
```

Results: Table 4.8 lists the results for version 1.6 of the pmp program using three different configurations of the operating system.

Table 4.8. Programmable Message Passing: Adjacent Pair

Period (ms)	Rate (MB/s)	OS	Coprocessor
5.0	52.4	OSF/1 R1.2.1	yes
7.1	36.9	OSF/1 R1.2.1	no
2.0	131.1	SUNMOS	no

Discussion: The message passing pattern given above is the same one generated by the adjacent pair benchmark with acknowledgments, however, measured throughputs are smaller. A variety of factors contribute to the differences. The most significant difference is that the rate calculated in the previous adjacent pairs benchmark is averaged over 1000 messages, while the rate in Table 4.8 is determined from the minimum period set by the worst case individual message time, which can be significantly impacted by a single operating system drop out. There is also more overhead in both the control and message passing parts of the pmp program. A final possibility is that each program aligns message buffers differently, and buffers not aligned on page boundaries can result in significant performance losses.

4.3.3 Corner Turning on a Line

The Programmable Message Passing Benchmark was developed to make it easier to evaluate complicated message passing patterns. In this section we apply the pmp program to the problem of corner turning on a line. Corner turning is the name given for transposing a matrix in some signal processing applications. In Section 6 scalable real-time mappings are considered in which the matrices involved are distributed across many nodes. The corner-turning step then consists of three phases as shown in Figure 4.6. First there are memory transposes or "corner turns" at the source nodes that take the matrix stored by rows, say, and restore it by columns. This is required if the information is to be packaged into large messages for more efficient transmission. The second phase corresponds to the data distribution phase shown in step 2 of Figure 4.6. Finally, a second memory corner turn at the destination nodes is needed to unpack the messages and to store the result by columns for subsequent processing. In this section we focus on the data redistribution at step 2.

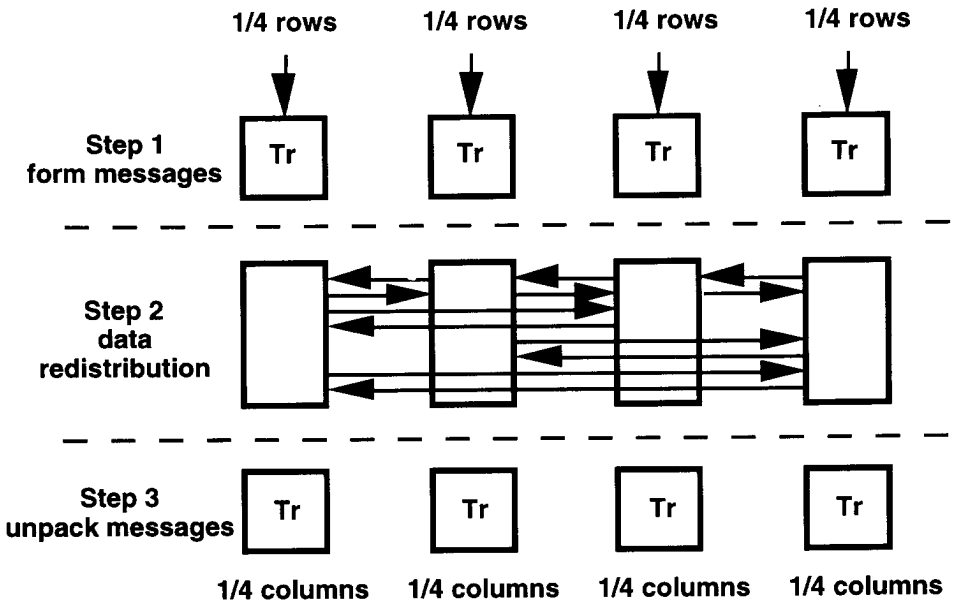


Figure 4.6. Corner Turning on a Line

Description: Figure 4.7 shows an example of pmp program input code sequences that implement the corner-turn data redistribution on a line of four nodes. Individual messages all have size 1/4 MB. The figure shows a graphical interpretation of the message passing program as occurring in six stages. This is somewhat misleading. Although the starts of the code sequence processing by pmp are synchronized across the nodes, there are no additional explicit synchronizations performed during the running of pmp to keep the messages aligned. However, because the messages are equally sized, one might expect that the stages shown in Figure 4.7 would be roughly correct.

pmp program input

```
0: R  S 3 262144  R  S 2 262144  S 1 262144  R  E
1: S 2 262144  R  S 3 262144  R  R  S 0 262144  E
2: R  S 1 262144  S 0 262144  R  S 3 262144  R  E
3: S 0 262144  R  R  S 1 262144  R  S 2 262144  E
```

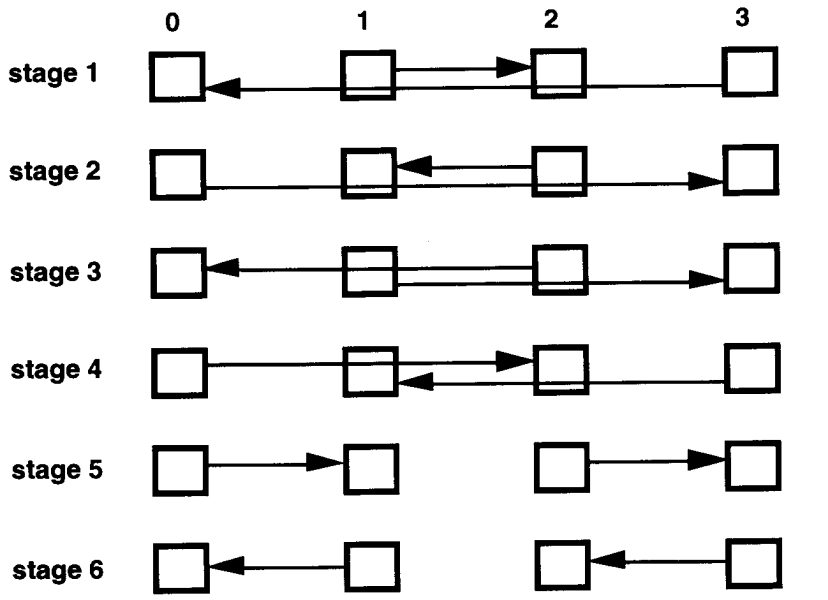


Figure 4.7. Corner-Turning pmp Code Sequences: Four Nodes

Results: Table 4.9 lists the results for version 1.6 of the pmp program using three different configurations of the operating system. The rates shown correspond to the total amount of data *sent* by any one node divided by the minimum period.

Table 4.9. Programmable Message Passing: Four-Node Corner Turning

Period (ms)	Rate (MB/s)	OS	Coprocessor
32.5	24.2	OSF/1 R1.2.1	yes
45.0	17.5	OSF/1 R1.2.1	no
44.0	17.9	SUNMOS	no

Discussion: Each node sends three 1/4 MB messages during the period, which accounts for the rates given in Table 4.9. But in that same period, each node receives three 1/4 MB messages. So the rates in Table 4.9 should be doubled if a fair comparison is to be made to the results of previous benchmarks in which nodes were either only sending *or* receiving, e.g., in comparing the results to those shown in Table 4.8. With this proviso, the corner-turning message passing performance obtained using OSF/1 appears to be consistent. But the performance under SUNMOS is disappointing.

One possible explanation for the relatively poor performance of SUNMOS is that the lack of actual synchrony between the stages in the pmp program may be more problematic for SUNMOS. Because SUNMOS does not break up the long 1/4 MB messages into small packets, any offsets between the stages of Figure 4.7 would result in serious contention, with the result of messages potentially piling up at the nodes. A synchronous version of pmp, which partitioned the period into subperiods (six in this case), would potentially avoid this problem.

There are, of course, a large number of alternative pmp code sequences that could be used to implement corner turning on a line of four nodes. Assuming the staging shown in Figure 4.7

is accurate, the pattern shown in the figure does not contain any contention at the nodes (no simultaneous sends and receives) nor on the backplane (messages are traveling in different directions). Other message patterns tested with either nodal or backplane contention did not perform as well under OSF/1 without the message coprocessor. Interestingly, with the message coprocessor turned on, the performance penalty for simultaneous sends and receives disappeared.

4.4 CONCLUSION

This section focused on pure communication benchmarks—no significant computation was performed at each node. The benchmarks determined upper bounds on message passing rates for the Intel Paragon. Lower bounds for the periods of repetitive real-time message passing were determined. Over the course of the year there was a general trend of improving performance as the hardware and operating system were upgraded (e.g., message coprocessor enabled) and as we discovered various "tricks" (e.g., explicit acknowledgments). SUNMOS provides very efficient use of the backplane, but shifts certain responsibilities to the user. High performance message passing at the rates that we have achieved can lead to potential predictability problems for a real-time application if communications between distance nodes are interfered with by messages involving intervening nodes. Also, as the result for the SUNMOS corner-turning benchmark suggests, more precise communications scheduling may be required to maintain high performance for complicated message passing patterns.

SECTION 5

APPLICATION BENCHMARKS

In this section we examine pipeline configurations of relevant processing and communication kernels on the Paragon to assess the degree to which high processing efficiencies can be maintained in end-to-end implementations. The applications under consideration are those in which problems present themselves at a fixed rate. Examples include synthetic aperture radar (SAR) image formation or space-time adaptive processing (STAP). To keep up with the stream of incoming data, the processor must meet a throughput requirement. The application may also prescribe a latency requirement.

This section demonstrates the use of double buffering to overlap communication and computation to maintain high processing efficiencies. Buffers must be appropriately sized to take into account a variety of factors. Large buffers usually result in more efficient processing or message passing as we have seen in the benchmarks in Sections 3 and 4. Small buffers may be required to meet strict latency requirements, as the time to fill a large buffer may be too great and result in timing violation. Finally, unpredictable operating system behavior will limit the size of the period that can be reliably maintained. This places a lower bound on how small the buffers can be.

5.1 REAL-TIME TEST BENCH

All benchmarks in this section are "real time" in the sense that the shortest period that can be sustained will be the metric measured. Ideally, given an application, a "test bench" would be constructed to realistically stimulate the massively parallel processor under test. This is the approach that the ARPA-sponsored Rapid Prototyping for Application Specific Signal Processors (RASSP) program is taking to assess the performance of the RASSP processors (Shaw, 1994). Instead, we construct what amounts to a "test bench" on the MPP itself.

In the case of applications that would use a single I/O node for input, we identify a single node, called the *source* node, that is responsible for injecting at a prespecified rate blocks of

data of a prespecified size into a processing chain under test. In reality, the data might stream into an I/O node, where it would be buffered to allow processing at the proper "grain" size. It is at this point that we begin our testing. The processing chain under test itself could be one- or two-dimensional.

If the output of the processing naturally coalesces at a single node, then we identify a single node, called the *sink* node, that is responsible for collecting the results for off-line verification of correct functionality. This simple test bench pair (source-sink) could be generalized to handle applications that naturally would have multichannel inputs or outputs. The major deficiency of this current set-up is that it does not assess the real-time I/O capabilities of the MPP under test. That will need to be done, especially in the context of actual applications that have nonstandard I/O interfaces.

The source node currently uses a "spin-wait" loop to determine when to send data, i.e., the clock is polled using the `dclock` routine and compared to a threshold. When the threshold is exceeded, corresponding to the beginning of a period, *and if there is a request for data pending*, then a data block is transmitted. Otherwise, the source node blocks until a request for data is received. After the data block has been sent, the source node checks to see if the deadline for that period was met. If a bottleneck occurs downstream, then the data request will be posted late, causing the source node to miss a deadline, which is noted. In this way the pipelines under consideration in this paper are self-timed. A synchronized pipeline could also easily be considered in the future on the Paragon because of the well-synchronized clocks at the processing nodes.

A key concern with this methodology is just how regularly can the data be injected. Operating system "drop outs" at crucial points in time affect the degree of regularity. The results of the clock benchmarks of Section 4.2 are relevant here. The conclusion for the current operating system (OSF/1) is that periods must be on the order of 100s of milliseconds if the current drop-out effects are to be ignored, i.e., have only a 1% -2% impact.

The test bench in this paper uses a single source node, so there is no need to synchronize the timers across multiple source nodes. A period and duration are selected at runtime, which

determine the number of iterations. Long durations account for the behavior of the operating system. Short durations that fall between operating system dropouts yield the best results and establish the potential for improvement by including a real-time operating system.

Most scientific codes running on MPPs are designed to be loaded homogeneously, i.e., the same code is loaded onto the processing nodes. This approach is well suited to data-parallel applications or applications in which the data is distributed across the nodes and the nodes cooperate to perform some computation on it. It is easy to synchronize the nodes under this model. If the nodes are performing different tasks, then the nodes dispatch to different subroutines that implement the proper functionality based on their logical node number. Individual nodes only execute a small portion of the common program. Homogeneous code is appealing from a software engineering standpoint because there is only a single program to maintain. Finally, this approach could support future fault-tolerant options since each node has the code to perform any of the required functions.

If memory is a scarce resource, then loading different code onto the nodes can make sense. Heterogeneous code is consistent with pipeline processing, where each node has a distinct task. In the present benchmarking context, it allows the test bench (timing, data source, and data sink nodes) to be separated from the processing nodes. The data source and data sink nodes are oblivious to the processing function under test. We have implemented both homogeneous and heterogeneous versions of the benchmarks without noticing much of a performance difference. The benchmark results that are reported here correspond to the standard homogeneous approach.

5.2 SINGLE NODE BENCHMARKS

In this section we take the realistic benchmarking philosophy a step further by including the "connective tissue" that will be common to most pipeline processing applications: buffering and flow control. We examine the case that the functions under test are implemented on a single node. The goal is to have a real-time benchmarking methodology that will produce results that can be relied on when these single node implementations are ganged together

using well specified communication patterns, e.g., linear or two-dimensional pipelines. We again pick the FFT to illustrate the approach, and, for simplicity, focus on a 512-point block length. Of course the approach can be generalized to other block lengths and different functions altogether.

Pipelining can be used to achieve high performance by overlapping the communication and processing through the use of double buffering. While iteration i is being computed on buffer B_1, the data for iteration $i + 1$ is being received into buffer B_2. If the computation takes at least as long as the time to receive the data into B_2, the communication overhead is hidden. When it comes time to compute iteration $i + 1$, the roles of the two buffers are reversed.

Eliminating data copies is also essential to achieve high performance, since data copying is an expensive operation, even compared to communication between processing nodes. Data copies occur when a message arrives at a node and there has been no corresponding receive request posted. Such messages must be buffered by the kernel in a system buffer and later copied into the application address space buffer. Messages that arrive for which a receive has been posted are copied directly into the application address space buffer and thus do not incur the cost of a copy from system to user space.

We now describe the buffering and flow control sequence that each processing node follows in the steady state of our benchmarks. During one iteration a processing node calls `irecv` to post a receive to fill buffer B_r with data for the next iteration. Then it sends a 0 byte message to the node upstream to request data for B_r. Ideally, the data will be received while the node computes its function on buffer B_c, which was filled during the last iteration. If the compute function is performed *in place*, then the results are also stored in B_c; otherwise the results of the computation are stored in a third (output) buffer B_o. After the function has completed on B_c, it sends the result, either from B_c or B_o, to the next node downstream *provided the compute node has already received a data request message from the next node*. Otherwise the node blocks until the request for data is received. After the send returns, the compute node will block if B_r has not been filled. The goal however is to require enough computation so that B_r is filled. If this is indeed the case, then the node does not block and proceeds immediately to the final step of swapping the pointers to B_r and B_c.

Figure 5.1 shows the three functions tested and the buffering used:

1. Applying a 512-point FFT to the rows of a 512×512 single precision complex matrix using the Kuck & Associates `cfft1d` call. This computation is performed in place.
2. Corner turning the 512×512 matrix within the memory of a node using the transpose function call `dtran`. This function is not done in place, and the result is stored in the output buffer.
3. A combination of function 1 followed by function 2 on the same node.

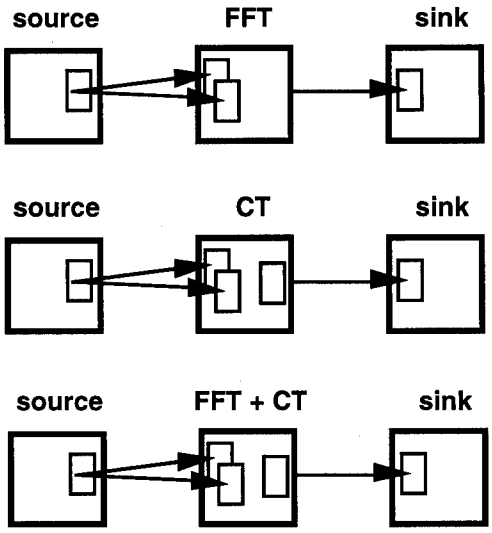


Figure 5.1. Single Node Processing Configurations

All benchmarks were run with OSF/1, revision 1.2.4, with the message coprocessor enabled on a 16-node Paragon at Honeywell. Table 5.1 lists the results obtained for a 4 second duration experiment. Two cases are compared:

1. Without data flow: In this case the compute nodes just switch between the two input buffers; there is no data sent from the source node and no results are sent to the sink node.
2. With data flow: In this case the source node repeatedly sends 512×512 single precision complex matrices ($2^9 \times 2^9 \times 2^3 = 2^{21} = 2$ MB messages) at the prescribed periods, and the results containing a similar amount of data are sent to the sink node.

Table 5.1. Single Node Processing Benchmark Results

	Without Data Flow		With Data Flow	
	Min Period (ms)	Rate	Min Period (ms)	Rate
FFT	205	57.5 MFLOPS	223	52.9 MFLOPS
CT	64	32.8 MB/s	109	19.2 MB/s
FFT+CT	267	44.2 MFLOPS	274	43.1 MFLOPS

Discussion: Comparing the results for the FFT with and without data flow reveals a difference in the minimum period that can be sustained of 18 ms. A periodic communication benchmark for 2 MB messages reveals that message passing alone requires a period of 26 ms (79 MB/s), less than 1/10 of the processing load. This would be approximately doubled to 52 ms for the receiving and sending that the processing node must do. Evidently about 2/3 of the communication overhead is hidden by double buffering. Further evidence of an overlap is implied by versions of the FFT benchmark that were run using only single input buffers. In this case, the difference in minimum periods is 37 ms. Additional benchmarks that turned off the receiving or sending at the processing node separately revealed that the original 18 ms were lost due to sends to the sink node and that the receiving communication was completely overlapped. A future task will be to modify the infrastructure to also overlap the sending phase to as great an extent possible.

The corner turn, which involves a memory copy to rearrange how the matrix is stored, takes about 1/3 the time to complete compared to the FFT computation. Combining the FFT and corner-turning function reduces the efficiency of the FFT processing, but with less impact when there is data flowing. In that case the communication overhead is almost completely hidden (only a 7 ms difference between the two cases with and without data flowing). This improvement could be a result of the third output buffer for this case.

5.3 PIPELINE BENCHMARKS

Arbitrary processing chains can be built by stringing together the source node, various processing modules, and the sink node. Figure 5.2 shows two pipelines that were tested. Both these pipelines implement two-dimensional FFT processing, which is applicable to SAR image formation for example. If the second FFT was replaced with an inner product calculation, then these pipelines would correspond to part of a pulse-doppler STAP application. The adaptive weight computation in the STAP application could be included in the corner-turn node of the longer pipeline as that node is lightly loaded. Table 5.2 lists the performance obtained for these pipelines when there is data flowing down the pipeline with the same parameters as used in Section 5.2.

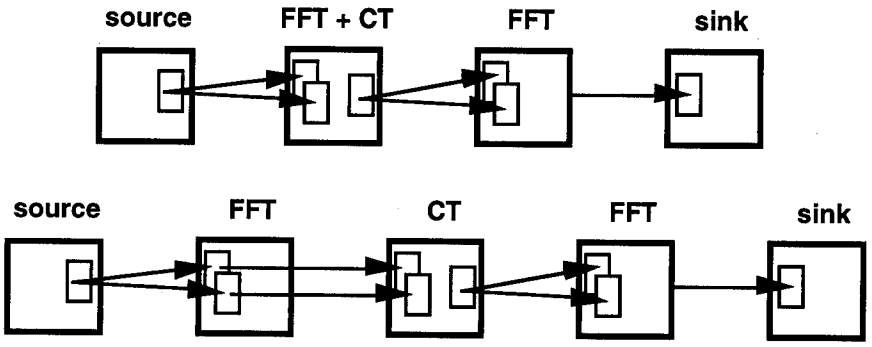


Figure 5.2. Pipeline Processing Configurations

Table 5.2. Pipeline Processing Benchmark Results

	Min Period (ms)	Rate
FFT+CT->FFT	274	43.1 MFLOPS
FFT->CT->FFT	223	52.9 MFLOPS

Discussion: As expected, the throughput of the pipeline is determined by the bottleneck node, i.e., the node with the largest minimum period. There is no further degradation in the

minimum period as a result of the longer pipeline. This result does mask one important effect that we have also measured. *Jitter* accumulates as we move down the pipeline. In other words, if the results are time stamped at various points in the pipeline, and differences computed at a single point for successive iterations, then the variability of the differences around the prespecified period increases as we move down the pipeline. This is a result of an accumulation of "nondeterministic" events occurring at each stage of this self-timed pipeline. One way to limit this effect would be to implement synchronized processing in which each node performs its function at prescribed intervals by using the `dclock` procedure to poll the synchronized (to within 1 μs) local clocks.

A key issue that was raised at the beginning of this section is the buffer size. This size is related to the number of instances of the incoming data stream that are coalesced and processed as a group. For example, in the pipelines considered above, the key parameter at the first stage is the number of rows that will be processed by the FFT. Such "granularity" studies that determine the appropriate buffer size as a function of all the relevant variables will be the subject of future work.

5.4 CONCLUSION

This section described benchmarks that are more predictive of actual application performance since they include both communication and computation. An infrastructure that implements a test bench on the Paragon itself was developed for use with a variety of processing functions. In particular, the test bench is used in conjunction with pipelined processing chains that represent functions relevant to SAR and STAP processing. The test bench is data-request driven to minimize data copies (all receives of large data messages are posted) and double buffered to hide communication latency. In our 512 x 512 example, approximately 2/3 of the communication latency is hidden. Even though the existing operating system is not real-time, by operating at a sufficiently coarse level of granularity, we are able to inject and process data at a regular rate. The benchmarks sustain 53 MFLOPS on applying FFTs to the rows and columns of a 512 x 512 matrix. This level of end-to-end processing efficiency is desired for embedded applications with strict size, weight, and power restrictions.

SECTION 6

SCALABLE REAL-TIME SYSTEMS

Parallel processing uses multiple processors to reduce the amount of time required to execute an algorithm, that is to *speed up* a computer program that implements the algorithm. The rate of reduction in the run time as additional processors are added is often regarded as a key descriptor of how effective parallel processing is for the particular application. A system's *scalability* refers to the nature of the progression of reduced run times as the number of processors is increased. For a fixed problem size, there is a limit to the speedup that can be obtained with some applications bottoming out quickly because they contain large proportions of unavoidably sequential operation (Amdahl's law). As a result, there have been alternative measures of scalability proposed that describe the behavior of the system as both the problem size and the number of processors are increased (Gustafson, 1988). In this section we discuss the issue of scalability for real-time parallel processing.

6.1 REAL-TIME SCALABILITY

Real-time systems must compute their results within prescribed timing requirements, which we have described in terms of latency and throughput requirements. One might be interested in how many processors are required to satisfy the constraints in the first place, which relate to the traditional notion of scalability. We focus on what must be done to continue to meet the real-time requirements when some dimension of the problem is increased. In this case we desire an extension of the mapping of the larger problem to, most likely, a larger number of processors so that the result continues to meet the real-time requirements. The amount of work required to obtain the extended mapping is a key concern, with parameterized solutions being preferred over mappings that require some fundamental shift in approach, e.g., replacing pipelining with data parallelism. Note that the real-time requirements themselves may also change as the problem size increases.

If the latency l is greater than the period p, then the computing system can be working on more than one problem instance at a time. If a processing node can satisfy the latency requirement, then a conceptually simple parallel processing approach for meeting *any* throughput requirement is called *replication*. In the replication approach the individual processing node is replicated at least l/p times with each copy receiving a successive problem instance. When the first processing node finishes the first problem instance, it becomes available to process the next available problem instance, and so on. As the problem size increases, the replication approach usually can be scaled to maintain the fixed throughput requirement (as long as the problem still fits), but the latency is increased.

Pipelining is another parallel processing approach that has been traditionally used in high throughput applications. The notion of a *scalable* dataflow graph was introduced in (Games, et al., 1994) to formalize the notion of scalability for real-time pipeline processing. The idea is to associate a parameterized family of dataflow graphs with the problem. The nodes of any graph in the family have a constant processing and communication load. This notion was based on the model of systolic array computational structures. If the graphs in the parameterized family can be mapped to the MPP with the constant workload property preserved at the nodes, then pipeline processing can be used to maintain a fixed throughput requirement, independent of problem size. Whether or not a fixed latency requirement continues to be met depends on whether the "depth" of the parameterized dataflow graph increases or not.

As the above discussion suggests, it is usually more difficult to maintain strict latency requirements as a problem size increases. In this section we develop these ideas further for a generic processing chain that is applicable to both SAR image formation and STAP applications. The goal is to obtain a scalable real-time implementation that maintains a fixed throughput *and* latency.

6.2 GENERIC PROCESSING CHAIN

Consider the following generic "two-dimensional" processing chain that generalizes the processing considered in Section 5.3:

1. the input is an $m \times n$ matrix A,
2. apply a function f to each column of A, creating an $m \times n$ matrix B, (for instance, the function f could be an FFT),
3. apply a function g to each row of B. (for instance, the function g could be an FFT or convolution in a SAR application or an inner product in a STAP application).

We are interested in examining the ramifications of scaling the above $m \times n$ input matrix in either one of its dimensions. Without loss of generality, we consider the case of scaling the parameter n. For example, if the columns of A represent the pulse returns of a SAR system, then scaling n would correspond to increasing the azimuth or cross-range resolution of the SAR image. In a STAP application, the columns could represent the pulse returns at a fixed range from a fixed antenna. Scaling n then would correspond to increasing the number of antennas, or spatial degrees of freedom. For simplicity, the three-dimensional STAP problem (pulses, antennas, and ranges) is reduced to the two-dimensional case we are considering by fixing the range dimension.

The generic processing chain can be implemented as a one-dimensional pipeline similar to the ones considered in Section 5. However the amount of work at the first node (n applications of the function f) of these mappings is not constant, and so the throughput cannot be maintained as n is increased. This suggests that the pipeline processing chain must also accommodate processors in a second dimension. That is, the input matrix must be partitioned and distributed across a line of processors to maintain a fixed workload per processor, independent of the size of n.

One potential mapping with this property is shown in Figure 6.1 for the case of four nodes on the line (four is only used for illustrative purposes). The stage 1 processing nodes receive 1/4 of the columns of the matrix. We do not consider the issue of input here, but pick up the processing at the input of stage 1. This mapping has a latency equal to three periods. The

corner turn at the second stage allows the second function g to be implemented on single nodes at the third stage, which is usually the most efficient processing approach. This corner-turn stage implements the three stages shown in Figure 4.6. The question that we consider next is: as the input parameter n is increased, can the mapping in Figure 6.1 be scaled to maintain a fixed throughput and latency constraint?

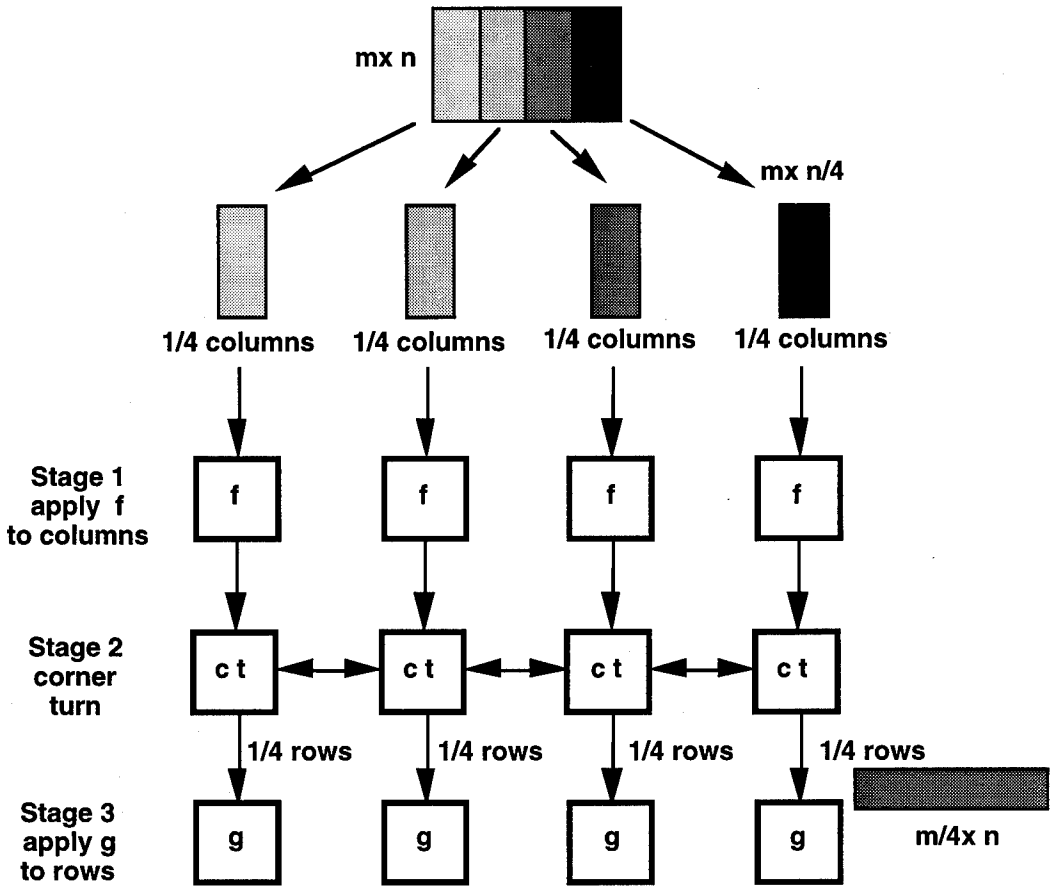


Figure 6.1. Two-Dimensional Distributed Mapping of Generic Processing Chain

6.3 SCALABLE REAL-TIME MAPPING

To precisely describe the scalability problem for the generic processing chain, we need to specify the parameterized family of mappings corresponding to increasing the input parameter n. For simplicity we restrict ourselves to the case where n is increased to $2n$, $4n$, $8n$, etc., and limit our discussion to the corresponding mapping for $2n$. The progression to higher values of n will then be clear. We will refer to the mapping for the value $2n$ as the *doubled* mapping.

The doubled mapping is obtained by doubling the number of nodes in each line of Figure 6.1. The input to the doubled mapping is a matrix of dimension $m \times 2n$. The input matrix is now divided into eight parts and distributed among the stage-1 nodes in a similar manner as in Figure 6.1 so that each stage-1 node still receives an $m \times n/4$ matrix. Thus the nodes at stage 1 of the doubled mapping perform precisely the same work as before: $n/4$ applications of the function f to the columns of size m. Thus the throughput into stage 1 can be maintained. However, the stage-2 corner turn involves twice as many nodes, and the stage-3 nodes process longer inputs (in particular the rows of an $m/8 \times 2n$ matrix). The performance of these latter two stages must be examined to determine if the doubled mapping still meets the real-time requirements.

Consider first the processing at stage 3. The amount of data involved in the doubled mapping at this stage is the same—$mn/4$ elements—so there will be no problem with memory. The issue reduces to the relative times of computing $m/4$ applications of the function g to an n-point input versus $m/8$ applications of g to a $2n$-point input. This obviously depends primarily on the function g. For example, for an FFT with operation count given by equation 3-1, the ratio of floating point operations required for the doubled ($2n$) mapping versus the single (n) mapping at stage 3 is $1 + 1/\log_2 n$. This suggests for the FFT that the stage-3 nodes in the doubled mapping will be able to complete their processing in time as long as there is only a small amount of margin in stage 3 of the single mapping. A number of issues could affect the ultimate performance, for example, the longer input length may not be accommodated in cache. Periodic benchmarking would provide the definitive answer.

The situation for the corner turn at stage 2 is less clear. Again the amount of data involved at each stage-2 node is the same in both the single and doubled mapping. But the number of total messages increases quadratically, from 12 to 56 when n doubles from 4 to 8. But each message is half the size, and the amount of data sent and received at each node increases only slightly (from 3/4 to 7/8 of its data when n doubles from 4 to 8). Thus, the resulting period that can be sustained by the corner-turning stage will depend on a number of factors, including the amount of concurrency in the larger message-passing pattern (positive factor) as well as the drop-off in efficiency as the message size decreases (negative factor).

The Programmable Message Passing Benchmark, introduced in Section 4, can be used to determine the minimum period for corner turning as a function of line size. Again, to make this precise, we need to describe the message-passing pattern to be used as the number of nodes is doubled: $n = 2, 4, 8, ..., 2^i, ...$ We give a construction of a parameterized family of corner-turning message-passing patterns based on two primitives called *swap* and *quad*. Figure 6.2 shows the swap(i, j) and quad(i, j) primitives for nodes labeled i, $i + 1$, j, and $j + 1$. In each primitive there is a succession of stages with the property that at each stage every node is sending or receiving exactly one message, i.e., there is no contention at the nodes.

When there are two nodes {0, 1}, they swap messages: swap(0, 1). The 4-node corner-turning pattern shown in Figure 4.7 corresponds to a quad(0, 2) followed by a simultaneous application of swap(0, 1) and swap(2, 3). An 8-node corner-turning pattern can be constructed in three steps by:

1) simultaneous applications of quad(0, 4) and quad(2, 6); followed by
2) simultaneous applications of quad(0, 6) and quad(2, 4); and finally
3) simultaneous applications of the 4-node corner-turning pattern to each half: {0, 1, 2, 3} and {4, 5, 6, 7}.

These three steps are shown schematically in Figure 6.3.

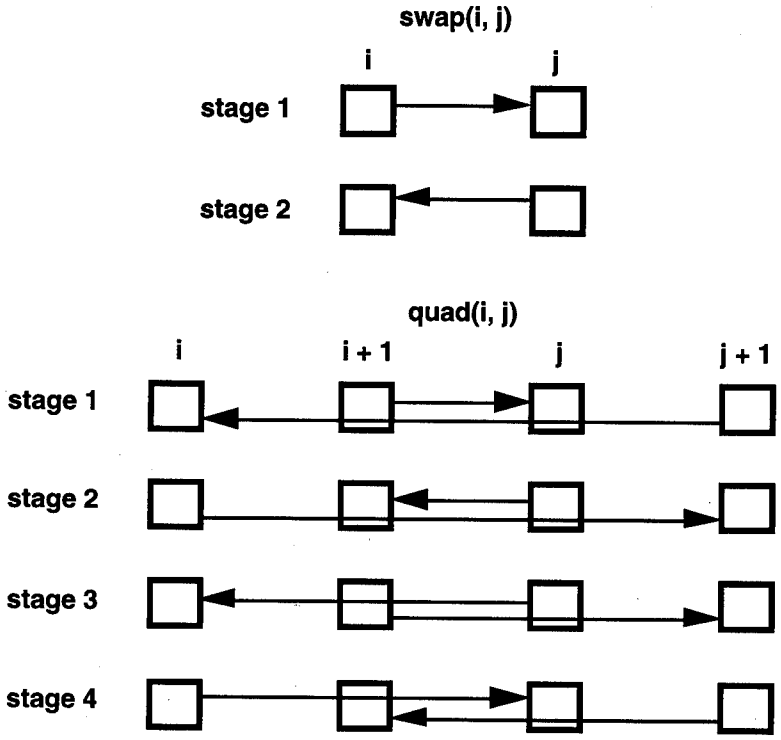


Figure 6.2. Swap and Quad Primitives

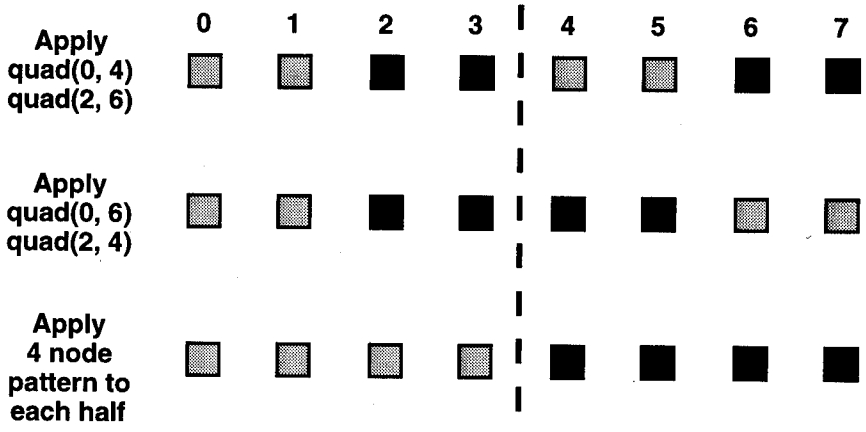


Figure 6.3. Eight-Node Corner-Turning Pattern Schematic

We now give the formulas for the general case. Given n processors, n even, and a parameter s, called the *skip* factor, we define a corner-turning step pattern, written step(n, s), as the simultaneous application of

$$\begin{gathered}\text{quad}(0, n/2 + s \bmod n/2) \\ \text{quad}(2, n/2 + (s+2) \bmod n/2) \\ \vdots \\ \text{quad}(n/2 - 2, n/2 + (s + n/2 - 2) \bmod n/2).\end{gathered}$$

In this notation, the 8-node corner-turning pattern corresponds to step(8, 0), then step(8, 2), and finally the corner-turning pattern for four nodes applied to the two halves. The 16-node corner-turning pattern can be constructed as a sequence of steps: step(16, 0), step(16, 2), step(16, 4), step(16, 6), and finally the corner-turning pattern for 8 nodes applied to the two halves. When the number of processors $n = 2^k$, the corner-turning pattern consists of step(n, 0), step(n, 2), ..., step(n, $n/2 - 2$), and finally the corner-turning pattern for $n/2$ nodes applied to the two halves.

This construction has no contention at the nodes: each stage has $n/2$ disjoint send-receive pairs. But there is contention on the links for $n \geq 8$, and this contention becomes exponentially worse as n is increased to 16, 32, etc. This will lead to problems of uneven communication rates due to the failure of fairness phenomenon that was previously illustrated in the Many Pairs Benchmark of Section 3. This suggests a future problem of developing a corner-turning construction that preserves the no-node contention property for the $n/2$ send-receive pairs with a slower growing link contention problem. A counting argument for the middle link, the worst case, reveals that contention for it must grow at least linearly (proportional to $n/8$) as the number of nodes increases. This problem is particularly relevant for the Paragon, because its $16 \times n$ layout results in possibly large values of n. On the other hand, size, weight, and power requirements of embedded applications would tend to limit the size of n.

The recursive construction for corner-turning patterns was implemented by a small Scheme program that generates pmp input code sequences. Scheme (IEEE, 1991) is a lexically scoped dialect of Lisp and is good for manipulating programs as data. Figure 6.4 shows the resulting input code sequences for $n = 8$ in an abbreviated form using the fact that all the messages have the same size, i.e., **S N_0 N_1** is written as **SN_0**. We assume that 1/4 MB messages are sent when $n = 4$, so that for $n = 8$ the messages have size 1/8 MB.

pmp program input
(messages size 1/8 MB)

```
0: R S7 R S6 R S5 R S4 R S3 R S2 S1 R E
1: S6 R S7 R S4 R S5 R S2 R S3 R R S0 E
2: R S5 R S4 R S7 R S6 R S1 S0 R S3 R E
3: S4 R S5 R S6 R S7 R S0 R R S1 R S2 E
4: R S3 S2 R R S1 S0 R R S7 R S6 S5 R E
5: S2 R R S3 S0 R R S1 S6 R S7 R R S4 E
6: R S1 S0 R R S3 S2 R R S5 S4 R S7 R E
7: S0 R R S1 S2 R R S3 S4 R R S5 R S6 E
```

Figure 6.4. Corner-Turning pmp Code Sequences: Eight Nodes

The benchmark was run using pmp, version 1.6 of 30 August 1994 under OSF/1, revision 1.2.4, with the message coprocessor enabled. The duration was set at 5 seconds. The results for lines with 2, 4, 8, 16 nodes are given in Table 6.1. Note there is a slight degradation for $n = 4$ (from 24.2 MB/s reported in Table 4.10 to 20.2 MB/s) because of the strong order cache patch.

Table 6.1. Programmable Message Passing: *n*-Node Corner Turning

n	Period (ms)	Rate (MB/s)	Message Size
2	19	27.6	1/2 MB
4	39	20.2	1/4 MB
8	46	19.9	1/8 MB
16	67	14.7	1/16 MB

Discussion: It is clear from these results that the corner-turning implementation is not a scalable mapping. But what is also clear is that the rate of increase in the achievable real-time period is *not* dramatic, i.e., even though the number of nodes doubles in each case, the period does not. Thus, although the corner-turning stage would most likely ultimately limit the real-time scalability of this three-stage mapping, the sublinear increase in the rate implies it should be possible to meet the real-time requirements over a usefully large range of values of n. The key issue of course would be how the corner-turning period, including the overhead of the intranode memory corner turns, compared to the periods of applying the functions f or g. At the granularity considered in this paper, the function periods leave significant room to make the line considerably longer. Note that the intranode memory corner turns could be shifted among the processing stages if that proved useful in terms of load balancing. Also, stages of the corner turn could be split among a number of pipeline stages if need be to maintain a fixed throughput at the cost of increased latency.

6.4 CONCLUSION

This discussion serves to sharply focus the benchmarks required to establish the scalability of a particular instance of the generic processing chain. The granularity (block size) at the first stage would be determined by the throughput requirement and the performance of the function f. The mapping is designed so that the processing and communication load involving this first stage is independent of problem size. The key issue at the corner-turn stage is determining the available margin that will limit the increase of line size, and hence problem size. The key assessment for the function g is the comparison of the performance of g on varying size inputs. Functions whose implementation complexity doubles as the length of the input doubles will result in scalable real-time behavior.

SECTION 7

CONCLUSION

This paper investigated the application of commercial massively parallel processors to real-time sensor processing. The focus was on a benchmarking methodology that supports the development of parallel software for real-time applications. This parallel software development process will involve running benchmarks designed to assess the level of performance the MPP, or more precisely its components, can deliver on processing and communication kernels. These kernels will then be combined into a single implementation using a variety of parallelization approaches so that the application's timing requirements are satisfied. The process is expected to be iterative and will involve a series of benchmarks that incorporate increasingly more of the application's requirements. The approach adopted here is to make the initial benchmarking results highly predictive by including in the benchmarks much of the infrastructure and overhead required in a real-time implementation.

This led to the construction of a test bench on the MPP whose purpose was to realistically stimulate the processing or communication function under test. The benchmarking metric was the minimum period that could be sustained for a selected processing and/or communication kernel. This approach assesses the level of real-time performance provided by the current hardware/software system and can be used to chart the impact of including a real-time operating system in the future. To bound the real-time processing problem, the scope of this effort was limited to the front-end signal processing found in such applications as synthetic aperture radar (SAR) processing and space-time adaptive processing (STAP). Real-time processing such as automatic target recognition or tracking that includes data-dependencies will be considered in the future.

The benchmarks were applied to the Intel Paragon in anticipation of the availability of the Embedded Touchstone. Our experiences showed that optimized library routines allowed efficient processing at the nodes of the MPP. In the important case of FFT processing, the library routines were most efficient at transform lengths commensurate with the size of on-chip cache. There is a rather severe penalty paid for short length transforms (8, 16, and 32

points) due to modern pipeline microprocessor architectures. Unfortunately, the applications we are interested in can involve these small transform lengths. To overcome this problem, we developed an FFT library call that keeps the floating-point pipelines filled by completing many short FFTs for each call. This resulted in a dramatic improvement in the processing efficiency (from 18 to 78 MFLOPS for 8-point transforms).

Improving message-passing primitives for the Paragon allowed efficient communication on the backplane. The message-passing rates we were able to sustain improved by a factor of four over the span of the year (from roughly 20 to 80 MB/s). The message-passing rates under SUNMOS were higher still (150 MB/s). At these high rates, however, problems with messages interfering with each other become more significant. We demonstrated how the communication performance of two distance nodes was affected by the behavior of intermediate nodes, motivating the need for some type of communications scheduling to obtain predictable performance in real-time applications (Games, et al., 1994b).

We showed that processing efficiencies could be maintained for pipeline processing in which double buffering was used to largely hide the overhead of interprocessor communication. The real-time scalability of parallel implementations was considered in which the key consideration was the maintenance of the real-time requirements as the problem size is increased. A scalable real-time mapping of a generic two-dimensional processing chain applicable to SAR and STAP applications was developed and analyzed. The mapping requires data redistribution along a line of processors to implement a "corner turn." A message-passing pattern with sublinear growth was demonstrated for this step, which increases the practicality of the proposed mapping.

We plan to apply these ideas in the future to actual real-time implementations, starting with the Lincoln Laboratory Advanced Detection Technology Sensor (ADTS). The ADTS system is a Ka-band SAR sensor that the ARPA RASSP program is using as a benchmark (Shaw, 1994). This system has a throughput requirement determined by a maximum pulse rate of 556 per second. The RASSP benchmark has a latency requirement of 3 seconds. This latency requirement will motivate additional enhancements to our benchmarking methodology. In this paper, all the results involved fairly coarse grain processing (e.g., 512 FFTs of size 512

points). However such coarse grain processing is incompatible with strict latency requirements. A key consideration will be determining what level of granularity the MPP reliably can support in a real-time application. The shorter periods involved in finer grain processing amplify the problems associated with nonreal-time system software. It is anticipated that such granularity studies will be a very effective way to measure the impact of improving real-time system software.

LIST OF REFERENCES

Blitzer, F., 1993, "Militarized Touchstone Program," *Proceedings of the 1993 IEEE National Aerospace and Electronics Conference*, Vol. 1, Dayton, OH, pp. 137–143.

Brigham, E. O., 1974, *The Fast Fourier Transform*, ISBN 0-13-307496-X, Prentice Hall, Inc., Englewood Cliffs, NJ.

Dowd, K.,1993, *High Performance Computing*, O'Reilly & Associates, Inc., Sebastopol, CA, ISBN 1-56592-032-5.

Games, R. A., J. D. Ramsdell, J. J. Rushanan, 1994a, "Real-Time Parallel Processing Using Scalable Dataflow Graphs," preprint.

Games, R. A., A. Kanevsky, P. C. Krupp, L. G. Monk, 1994b, *Real-Time Embedded High Performance Computing: Communications Scheduling*, MTR 94B146, The MITRE Corporation, Bedford, MA.

Games, R. A., and D. S. Pyrik, 1994, *Parallel Implementation of the Planar Subarray Processing Algorithm*, MTR 94B114, The MITRE Corporation, Bedford, MA.

Golub, G. H., and C. F. Van Loan, 1991, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD.

Gustafson, J. L., 1988, "Reevaluating Amdahl's Law," *Communications of the ACM*, Vol. 31, No. 5, pp. 532–533.

IEEE Std 1178-1990, 1991, *IEEE Standard for the {Scheme} Programming Language*, Institute of Electrical and Electronic Engineers, Inc., New York, NY.

Intel, 1990, *i860 64-Bit Microprocessor Hardware Reference Manual*, Intel Corporation, Mt. Prospect, IL.

Intel, 1991, *i860 XP Microprocessor Data Book*, Intel Corporation, Mt. Prospect, IL.

Intel, 1992a, *i860 XP 64-Bit Microprocessor Hardware Reference Manual*, Intel Corporation, Mt. Prospect, IL.

Intel, 1992b, *i860 Microprocessor Family Programmer's Reference Manual*, Intel Corporation, Mt. Prospect, IL.

Kamenoff, N. I., and N. H. Wiederman, 1991, Hartstone Distributed Benchmark: Requirements and Definitions, *Proceedings of the 1991 Real-Time Systems Symposium, San Antonio, TX, 4-6 November 1991*, 0-8186-2450-7/91, Institute of Electrical and Electronic Engineers, Inc., New York, NY.

Kuck & Associates, 1993a, *CLASSPACK Signal Processing Library/C User's Guide*, Document #9310009, Release 1.3, Kuck & Associates, Inc., Champaign, IL.

Kuck & Associates, 1993b, *CLASSPACK Basic Math Library/C User's Guide*, Document #9311009, Release 1.3, Kuck & Associates, Inc., Champaign, IL.

Margulis, N., 1990, *i860 Microprocessor Architecture*, ISBN 0-07-881645-9, Osborne McGraw-Hill, Berkeley, CA.

Perry, R. P., R. C. DiPietro, A. Kozma, J. J. Vaccaro, April 1994a, "SAR Image Formation Processing Using Planar Subarrays," *Proceedings of the SPIE OE/Aerospace Sensing, Conference on Algorithms for Synthetic Aperture Radar Imagery*, Orlando, FL.

Perry, R. P., R. C. DiPietro, A. Kozma, J. J. Vaccaro, September 1994b, *SAR Image Formation Processing Using Planar Subarrays*, MTR 94B39, The MITRE Corporation, Bedford, MA.

Shaw, G. A., 1994, "RASSP Benchmark Program Overview," *Proceedings of the 1st Annual RASSP Conference*, ARPA, pp. 33–42.

SKY, August 1993, *SKYvec Software, Release 3.5 for Sun4 Hosts and Targets*, SKY Computer, Inc., Chelmsford, MA.

Weiderman, N. H., and N. I. Kamenoff, 1992, "Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems," *Journal of Real-Time Systems*, Vol. 4, pp. 353–382.

Wheat, S. R., R. Riesen, A. B. Maccabe, D. W. van Dresser, T. M. Stallcup, 1994, "PUMA: An Operating System for Massively Parallel Systems," *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, pp. 56–65.

Zima, H., and B. Chapman, 1990, *Supercompilers for Parallel and Vector Computers,* ACM Press, New York, NY.

Rome Laboratory

Customer Satisfaction Survey

RL-TR-_____

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

_____

_____

Organization Name:_____(Optional)

Organization POC: _____(Optional)

Address:_____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful 1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No_____

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

*U.S. GOVERNMENT PRINTING OFFICE: 1995-710-126-20055

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.